

The Transactor

The Tech/News Journal For Commodore Computers

95% Advertising Free!

July 1986: Volume 7, Issue 01. \$3.50

ROM Routines / Kernel Routines



Good news!

If you want to get the most out of your Commodore 128 or 64, we have good news for you. The Paperback 128 and 64 Series of Software both offer you serious, professional quality software packages that are easy to use and inexpensive.

How easy?

Paperback 128 or 64 Software is so easy, you're ready to start using it as soon as it's loaded into memory. Even if you've never been in front of a computer before, you'll be up and running in thirty minutes. In fact, you probably won't ever need the reference guide... 'help' is available at the touch of a key. That's how easy.

How serious?

Paperback 128 or 64 packages have all the power you're ever likely to need. They have all of the features you'd expect in top-of-the-line software, and then some. The good news is that Paperback 128 or 64 Software Packages are priced way down there... where you can afford them.

Fast, powerful, easy to learn and inexpensive. Say, that is good news!

All for one and one for all

Paperback 128 or 64 Software Packages offer you something else you might not expect... integration. You can combine the output of Paperback Writer, Paperback Filer and Paperback Planner into one piece of work. You can create a finished document with graphs, then send individually addressed copies.

The bottom line is Solutions

The word solutions is our middle name and bottom line. When you purchase Paperback 128 or 64 software, you can count on it to solve your problems.



For information write to:

Digital Solutions
P.O. Box 345, Station A
Willowdale, Ontario
M2N 5S9

Paperback Writer 128 or 64 Word Processing

What you see is what you get

With Paperback Writer 128 or 64, there's no more guessing what text will look like when you print it. What you see is what you get... on screen and in print. There are no fancy codes to memorize, no broken words at the end of a line.

Easy to learn and sophisticated. Paperback Writer 128 or 64 offers standard word processing features plus...

- on-screen formatting and wordwrap
- on-screen **boldface**, underlines and *italics*
- no complicated format commands to clutter text
- on-screen help at all levels
- spelling-checker lets you add words to your dictionary
- 40 or 80 columns on screen
- files compatible with PaperClip™ or other word processors

Paperback Planner 128 or 64 Computerized Spreadsheet

Make fast work of budgeting and forecasting

Paperback Planner 128 or 64 software lets you make fast work of all your bookkeeping chores. Cheque books, household accounts, business forecasting and bookkeeping are just some of the jobs that Paperback Planner 128 or 64 packages make easier. You can even create four different kinds of graphs.

Accurate, sophisticated and easy to use. Paperback Planner 128 or 64 offers standard spreadsheet features plus...

- accuracy up to 16 digits, about twice as many as most spreadsheets for the Commodore 128 or 64
- sideways printing available on dot matrix printers, for oversized spreadsheets that won't fit on standard paper
- **on-screen help** at all levels
- compatible with VisiCalc™ files
- 80 column on-screen option for the Commodore 64 in addition to the standard 40 columns
- graphics include **bar**, **stacked bar**, **line** and **pie** graphs that can also be used in word processing files
- **smart evaluation** of formulae for accurate complex matrices

Paperback Filer 128 or 64 Database Manager

Database management made easy

With Paperback Filer 128 or 64, you can organize mailing lists, addresses, inventories, telephone numbers, recipes and other information in an easily accessible form. Use it with Paperback Writer 128 or 64 (or other word processors) to construct individually customized form letters.

Paperback Filer 128 or 64 packages are fast, sophisticated and truly easy to use. In addition to standard database features they offer...

- use up to 255 fields per record (2,000 characters per record)
- sorts by up to 9 criteria, can save 9 different sorts
- print **labels** in multiple columns
- flexible report formatting including **headers** and **footers**
- optional password protection including **limited access viewing** or **updating**
- on-screen help at all levels
- print from any record to any record
- arithmetic and trigonometric functions in **reports** using up to 16 digit accuracy

Solutions!



**Paperback
Writer 64**
Word Processor

PW 128/64 Dictionary
also available at \$14.95 (U.S.)



**Paperback
Writer 128**
Word Processor

MAIL ORDERS:

CRYSTAL COMPUTER INC.
In Michigan 1-517-224-7667
Outside Michigan 1-800-245-7316
24 hours a day, 7 days a week.



**Paperback
Planner 64**
Spread Sheet



**Paperback
Filer 64**
Database



**Paperback
Filer 128**
Database



**Paperback
Planner 128**
Spread Sheet

Best-selling software for Your Commodore 128 or 64

You want the very best software you can find for your Commodore 128 or 64, right?

You want integrated software — word processing, database and spreadsheet applications — at a sensible price. But, you also want top-of-the-line features. Well, our Paperback 128/64 software goes one better.

With Paperback 128 or 64, you'll find all the features you can imagine ... and then some. And Paperback 128/64 is so easy to use, you won't even need the reference guide. On-screen and in memory instructions will have you up and running in less than 30 minutes, even if you've never used a computer before.

The price? It's as low as you'd expect for a line of software called 'Paperback'. Suggested Retail Price for the 64 Software is \$39.95 (U.S.) and \$49.95 (U.S.) for the 128. Any of the 64 products may be upgraded to their 128 version for \$15.00 (U.S.) + \$3.00 shipping and handling. (Available to registered owners from Digital Solutions Inc. only.)

Paperback Writer 128 or 64, Paperback Planner 128 or 64 and Paperback Filer 128 or 64 ... **Solutions** at sensible prices from Digital Solutions Inc.

International & Distributor enquiries to:

**Serious software
that's simple to use.**



P.O. Box 345, Station A
Willowdale, Ontario
Canada M2N 5S9
1-416-221-3225

Paperback Writer 128 and 64 are now available in French.

JOIN TPUG

The largest Commodore Users Group

Benefit from:

Access to library of public domain software
for C-64, VIC 20 and PET/CBM

Magazine (10 per year) with advice from

Jim Butterfield
Brad Bjomdahl
Liz Deal

TPUG yearly memberships:

Regular member (attends meetings)	— \$35.00 Cdn.
Student member (full-time, attends meetings)	— \$25.00 Cdn.
Associate (Canada)	— \$25.00 Cdn.
Associate (U.S.A.)	— \$25.00 U.S.
	— \$30.00 Cdn.
Associate (Overseas — sea mail)	— \$35.00 U.S.
Associate (Overseas — air mail)	— \$45.00 U.S.

FOR FURTHER INFORMATION:

Send \$1.00 for an information catalogue
(tell us which machine you use!)

To: TPUG INC.
DEPT. A,
101 DUNCAN MILL RD., SUITE G7
DON MILLS, ONTARIO
CANADA M3B 1Z3

COMAL INFO If you have COMAL— We have INFORMATION.

BOOKS:

- COMAL From A To Z, \$6.95
- COMAL Workbook, \$6.95
- Commodore 64 Graphics With COMAL, \$14.95
- COMAL Handbook, \$18.95
- Beginning COMAL, \$22.95
- Structured Programming With COMAL, \$26.95
- Foundations With COMAL, \$19.95
- Cartridge Graphics and Sound, \$9.95
- Captain COMAL Gets Organized, \$19.95
- Graphics Primer, \$19.95
- COMAL 2.0 Packages, \$19.95
- Library of Functions and Procedures, \$19.95

OTHER:

- COMAL TODAY subscription, 6 issues, \$14.95
- COMAL 014, Cheatsheet Keyboard Overlay, \$3.95
- COMAL Starter Kit (3 disks, 1 book), \$29.95
- 19 Different COMAL Disks only \$94.05
- Deluxe COMAL Cartridge Package, \$128.95
(includes 2 books, 2 disks, and cartridge)

ORDER NOW:

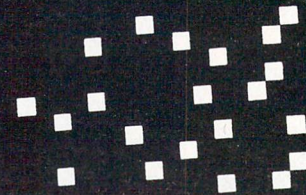
Call TOLL-FREE: 1-800-356-5324 ext 1307 VISA or MasterCard
ORDERS ONLY. Questions and Information must call our
Info Line: 608-222-4432. All orders prepaid only—no C.O.D.
Add \$2 per book shipping. Send a SASE for FREE Info
Package or send check or money order in US Dollars to:

COMAL USERS GROUP, U.S.A., LIMITED
5501 Groveland Ter., Madison, WI 53716

TRADEMARKS: Commodore 64 of Commodore Electronics Ltd.;
Captain COMAL of COMAL Users Group, U.S.A., Ltd.

Commodore Reference Diary

Jim Butterfield



1986

COMPUTER DIARY

From The Guru Himself! The 1986 Commodore Reference Diary

A 65 page reference section that includes:

- All hardware specifications including the C128 and PC10/20
- Useful memory locations
- Useful programs
- SuperCharts
- BASIC and machine language hints
- Hexadecimal conversion
- Sound, video
- and more

The full calendar and date book includes:

- National holidays in ten countries
- Personal notes
- 1987 forward planner
- Name, address, telephone section

Just \$5.95

(plus 50¢ postage and handling)

Order Your Copy Today!

Canada
The Transactor
500 Steeles Avenue
Milton, Ontario
L9T 3P7

USA
The Transactor
277 Linwood Avenue
Buffalo, New York
14209

Dealer Orders:

Canada
Norland Agencies
251 Nipissing Road
Milton, Ontario
L9T 4T7
(416) 876-4774

USA
MicroPace Distributing
1510 North Neil Street
Champaign, Illinois
61820
1 800 362-9653

Volume 7 Issue 01

Circulation 64,000

The Transactor

ROM Routines / Kernel Routines

Start Address Editorial 3

Bits and Pieces 5

1541 Error Allocator
Coloured Remarks Without REM
Directory Filename Highlighter
Easy Speedup For The C-128 With 1541 Drive
C-128 Key Repeat
C-64 Load It My Place Not Yours!
C-64 *Italics*
The SCNKEY Kernal Routine
C-64 and VIC Un-NEWS
Fast Memory Clear Using Garbage
(PRINT AT Update) Update
User Friendly Commands
REM RAM: Tag-Along Program Variables
Strings
Rhetorical Loops
SYSing With The C-128
Running ABasiC From The CLI On The Amiga
C-64 Auto-Start
The C-64 Great Escape
Return of The Swords Of Doom
Date Conventions
The Hidden Message
Verfizer For The Plus 4 and C128
C-128's Help Key Redefined
Amiga Lattice C Notes
Reading 8250-Formatted Disks with an 8050
1571's Can Be *too* Smart
Holy Input-Buffer, Batman!

Letters 13

Help: Line Scanner Required
Attention Hot 1541 Owners
"... but if you fool me twice then I'm indeed a fool."
Relative File Access In ML
Real Programmers... Verifier Update
Help Required Sky Travel Support
The Drive Disaster LADS to PAL Conversion

News BRK 77

Transactor on Microfiche
Transactor Mail Order News
The Viewtron Starter Kit
Transactor Open On Viewtron!
The Transactor Communications Disk
Music Workshops
Distressed Commodore Users Hotline
The 1541 Revealed
Used Computer Listing Service
Steve Jobs and Pixar Employees Buy Pixar
MSD Disk Drive Information Exchange
Introducing Super Kit/1541
DISKORGANIZER For The C64
Amiga Spreadsheet, Telecommunications and BBS
The Sourcerer 6500 Series Disassembler
Help Master For The Commodore 64
RESWITCH from Compusave
Uninterruptible Power Supply
80 Column Mono Cable For The C-128
40/80 Column Switch Cable For The C-128

TransBASIC Installment #9	18
Longer Life For The C64 and 1541	25
Matrix Manipulator ml array access techniques	26
Jim Butterfield's C128 Memory Maps	29
The C128 - You Can Bank On It	34
Getting The C128's CP/M In Gear	36
C128 RAM Disk a non-hardware project	38
AmigaBASIC Function Plot	42
Kernel Routines In The B128	44
Unmasking The Kernel	46
Kernel Who? more on the mysterious Kernel	48
Adding Functions To BASIC	54
Command Wedge intercepting BASIC commands	58
Improving The SYS Command	63
AutoLoad and the EPROM automatic load and run on power-up	66
SYMASS For The C64 a symbolic assembler that's PAL compatible!	69

**Note: Before entering programs,
see "Verfizer" on page 4 and 11**

Editor in Chief
Karl J. H. Hildon

Editor
Richard Evers

Technical Editor
Chris Zamara

Art Director
John Mostacci

Administration & Subscriptions
Lana Humphries

Contributing Writers

Ian Adam	Scott Maclean
Daniel Bingamon	Steve McCrystal
Neil Boyle	Jim McLaughlin
Anthony Bryant	Terry Montgomery
Tim Buist	Michael Mossman
Jim Butterfield	Gerald Neufeld
Gary Cobb	Noel Nyman
Jeffery Coons	Dave Pollack
Pierre Corriveau	Richard Perrit
Robert V. Davis	Terry Pridham
Elizabeth Deal	Raymond Quirling
Frank E. DiGioia	Glen Reesor
Yijun Ding	Gary Royal
Michael J. Erskine	John W. Ross
Jim Grubbs	John Russell
Tom Hall	Louis F. Sander
Bob Hayes	Fred Simon
John Jay Hilfiger	Perry Shultz
Andy Hochheimer	Edward Smeda
John Holtum	Darren J. Spruyt
Robert Huehn	Nick Sullivan
Tom Hughes	Zoltan Szepesi
Chris Johnson	Karel Vander Lugt
Mark Jordan	Audrys Vilkas
Clifton Karnes	Andrew Walduck
Gary Kiziak	Steven Walley
Jesse Knight	Jack Weaver
James E. LaPorte	Charles Whittner
William Levak	Evan Williams
Jack Lothian	Chris Wong

Production
Attic Typesetting Ltd.

Printing
Printed in Canada by
MacLean Hunter Printing

Program Listings In The Transactor

All programs listed in The Transactor will appear as they would on your screen in Upper/Lower case mode. To clarify two potential character mix-ups, zeroes will appear as '0' and the letter 'o' will of course be in lower case. Secondly, the lower case L ('l') has a flat top as opposed to the number 1 which has an angled top.

Many programs will contain reverse video characters that represent cursor movements, colours, or function keys. These will also be shown exactly as they would appear on your screen, but they're listed here for reference. Also remember: CTRL-q within quotes is identical to a Cursor Down, et al.

Occasionally programs will contain lines that show consecutive spaces. Often the number of spaces you insert will not be critical to correct operation of the program. When it is, the required number of spaces will be shown. For example:

print " flush right " - would be shown as - print "[10 spaces]flush right "

Cursor Characters For PET / CBM / VIC / 64

Down - q	Insert - T
Up - Q	Delete - t
Right - 	Clear Scrn - S
Left - [Lft]	Home - s
RVS - r	STOP - c
RVS Off - R	

Colour Characters For VIC / 64

Black - P	Orange - A
White - e	Brown - U
Red - L	Lt. Red - V
Cyan - [Cyn]	Grey 1 - W
Purple - [Pur]	Grey 2 - X
Green - t	Lt. Green - Y
Blue - +	Lt. Blue - Z
Yellow - [Yel]	Grey 3 - [Gr3]

Function Keys For VIC / 64

F1 - E	F5 - G
F2 - I	F6 - K
F3 - F	F7 - H
F4 - J	F8 - L

**Please Note: The Transactor has
a new phone number: (416) 878 8438**

The Transactor is published bi-monthly by Transactor Publishing Inc., 500 Steeles Avenue, Milton, Ontario, L9T 3P7. Canadian Second Class mail registration number **6342**. USPS **725-050**, Second Class postage paid at Buffalo, NY, for U.S. subscribers. U.S. Postmasters: send address changes to The Transactor, 277 Linwood Avenue, Buffalo, NY, 14209 **ISSN# 0827-2530**.

The Transactor is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names (PET, CBM, VIC, 64) are registered trademarks of Commodore Inc.

Subscriptions:
Canada \$15 Cdn. U.S.A. \$15 US. All other \$21 US.
Air Mail (Overseas only) \$40 US. (\$4.15 postage/issue)

Send all subscriptions to: The Transactor, Subscriptions Department, 500 Steeles Avenue, Milton, Ontario, Canada, L9T 3P7, 416 878 8438. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to Milton HQ. For best results, use postage paid card at center of magazine.

Editorial contributions are always welcome. Writers are encouraged to prepare material according to themes as shown in Editorial Schedule (see list near the end of this issue). Remuneration is \$40 per printed page. Preferred media is 1541, 2031, 4040, 8050, or 8250 diskettes with WordPro, WordCraft, Superscript, or SEQ text files. Program listings over 20 lines should be provided on disk or tape. Manuscripts should be typewritten, double spaced, with special characters or formats clearly marked. Photos or illustrations will be included with articles depending on quality. Authors submitting diskettes will receive the Transactor Disk for the issue containing their contribution.

Quantity Orders:

U.S.A. Distributor: Capital Distributing Charlton Building Derby, CT 06418 (203) 735 3381 (or your local wholesaler)	Master Media 261 Wyecroft Road Oakville, Ontario L6J 5B4 (416) 842 1555 (or your local wholesaler)	CompuLit PO Box 352 Port Coquitlam, BC V5C 4K6 604 941 7911	Norland Communications 251 Nipissing Road, Unit 3 Milton, Ontario L9T 4Z5 416 876 4774
--	---	---	--

SOLD OUT: The Best of The Transactor Volumes 1 & 2 & 3; Vol 4 Issues 04, 05, 06, Vol 5 Issues 03, 04
Still Available: Vol. 4: 01, 02, 03. Vol. 5: 01, 02, 04, 05, 06. Vol. 6: 01, 02, 03, 04, 05

Back Issues: \$4.50 each. Order all back issues from Milton HQ.

All material accepted becomes the property of The Transactor. All material is copyright by Transactor Publications Inc. Reproduction in any form without permission is in violation of applicable laws. Please re-confirm any permissions granted prior to this notice. Solicited material is accepted on an all rights basis only. Write to the Milton address for a writers package. The opinions expressed in contributed articles are not necessarily those of The Transactor. Although accuracy is a major objective, The Transactor cannot assume liability for errors in articles or programs. Programs listed in The Transactor are public domain; free to copy, not to sell.

S.N.I.F.F. : A Bold New Vision In Recording Media

Software with an added dimension of realism

Transactor magazine has just scooped the story on an incredible breakthrough in floppy-disk technology which promises to add a new dimension to software realism. Sensory Laboratories of Fremont Wyoming, a company which develops more natural man-machine interfaces, has been secretly developing their new floppy-disk technology, called "Sensory Nasal Interface For Floppies", or SNIFF.

The technology is now ready to be released for license by major software companies. Sensory Labs has managed to embed scents within the magnetic particles on a magnetic disk's surface. The basic idea works much like the "Scratch-n-Sniff" scent samples provided on paper carriers. The disk-based smells, however, are released by the heat-producing friction caused by the pressure pad opposite the Read/Write head. As the disk spins and the pressure pad rubs on the disk surface, the disk surface is slightly worn and heated, releasing the smells to the surrounding air, which is then wafted into the room through the drive's ventilation slots.

Software vendors should be excited by the new sniff-disks (known as "floppy-sniffs"), since they can add a realism to their programs which was never before possible. Sensory Labs' President, Terrence Price, explains: "The first computers printed all their results on paper. Then, we had the CRT, which eventually opened up the wonderful visual world of computer graphics. Now we have high-quality sound and speech synthesis as well. The sense of smell is the next logical step in human interface technology."

The first batch of disks will be released in 4 TPS (Tracks Per Sniff) format. On a 35-track disk like the Commodore 1541 uses, this will give eight different smells which can be released. (The directory tracks are not scented because they need to be accessed periodically during a sniff-access.) A program releases a desired scent by moving the Read/Write head to the proper sniff-track and holding it there for at least three seconds. (This is called a "Sniff access" or just a "Sniff".) Sniff-access time is expected to improve in future advancements of the technology. Applications are expected to include games (smell the musty dungeon in an adventure); and a whole range of Sniffware for the blind, who have a keener sense of smell and will be able to follow scent prompts from the programs.

Price admits that the idea is not completely original; he was inspired by the cinema technique known as Smell-o-Rama, most recently used in the movie "Polyester". But he maintains that Sensory Labs is going beyond simple one-at-a-time sniffs, into the exciting science of compound scent. Scientists at Sensory Labs have broken down smells into nine primary elements, out of which almost all other smells can be created. Smells can thus be synthesized from the primary smells on disk, as the Read/Write head quickly seeks from one track to another, blending the smells to create new ones.

"Once we come out with the 3 TPS format", explains Price, "We'll be able to put all of the primary smells on a floppy, making true sniff-synthesis a reality. At that stage, we can sell our 'Sniff-Writer' software which will allow developers or even users to create any smell they need without having to place a special order."

One of the problems being worked on still is the sniff-life of a disk; currently a typical track is good for about five sniff-hours. This may be enough for most games, but serious Sniffware will demand greater Sniff-lives. Improvements are on the way though, and Sensory Labs is even hoping to come out with a Hard-disk version called the "Hard-Sniff". Another potential problem is that in the event of certain hardware failures, the air in the room can be contaminated quickly. Sniff-disks come with warnings to use only in well-ventilated areas. This is especially true for programs using some of the stronger smells: for example, in an Adventure game the player may enter a recently-used bathroom.

Looking towards the future, Sensory Labs hopes to have 3.5 inch Sniff-disks out by June, and the Hard-sniff by next year. When asked about the future of Sniff-disk technology, Price predicts: "I see a major demand for Sniffware in the next few years, because people are always looking for new methods of getting information from their computers. And as we say here at Sensory Labs, 'a picture may be worth a thousand words, but a sniff is worth a million'".

And remember. . . you read it first in The Transactor April Edition - CZ

Using "VERIFIZER"

The Transactor's Foolproof Program Entry Method

VERIFIZER should be run before typing in any long program from the pages of The Transactor. It will let you check your work line by line as you enter the program, and catch frustrating typing errors. The VERIFIZER concept works by displaying a two-letter code for each program line which you can check against the corresponding code in the program listing.

There are two versions of VERIFIZER on this page; one is for the PET, the other for the VIC or 64. Enter the applicable program and RUN it. If you get the message, "***** data error *****", re-check the program and keep trying until all goes well. You should SAVE the program, since you'll want to use it every time you enter one of our programs. Once you've RUN the loader, remember to enter NEW to purge BASIC text space. Then turn VERIFIZER on with:

SYS 828 to enable the C64/VIC version (turn it off with SYS 831) or SYS 634 to enable the PET version (turn it off with SYS 637)

Once VERIFIZER is on, every time you press RETURN, on a program line a two-letter report code will appear on the top left of the screen in reverse field. Note that these letters are in uppercase and will appear as graphics characters unless you are in upper/lowercase mode (press shift/Commodore on C64/VIC).

Note: If a report code is missing it means we've edited that line at the last minute which changes the report code. However, this will only happen occasionally and only on REM statements.

With VERIFIZER on, just enter the program from the magazine normally, checking each report code after you press RETURN on a line. If the code doesn't match up with the letters printed in the box beside the listing, you can re-check and correct the line, then try again. If you wish, you can LIST a range of lines, then type RETURN over each in succession while checking the report codes as they appear. Once the program has been properly entered, be sure to turn VERIFIZER off with the SYS indicated above before you do anything else.

VERIFIZER will catch transposition errors (eg. POKE 52381,0 instead of POKE 53281,0), but ignores spaces, so you may add or omit spaces from the listed program at will (providing you don't split up keywords!). Standard keyword abbreviations (like nE instead of next) will not affect the VERIFIZER report code.

Technical info: VERIFIZER resides in the cassette buffer, so if you're using a datasette be aware that tape operations can be dangerous to its health. As far as compatibility with other utilities goes, VERIFIZER shouldn't cause any problems since it works through the BASIC warm-start link and jumps to the original destination of the link after it's finished. When disabled, it restores the link to its original contents.

Listing 1a: VERIFIZER for C64 and VIC-20

KE	10 rem* data loader for "verifier" *
JF	15 rem vic/64 version
LI	20 cs=0
BE	30 for i=828 to 958:read a:poke i,a
DH	40 cs=cs+a:next i
GK	50:
FH	60 if cs<>14755 then print "***** data error *****":end
KP	70 rem sys 828
AF	80 end
IN	100:
EC	1000 data 76, 74, 3, 165, 251, 141, 2, 3, 165
EP	1010 data 252, 141, 3, 3, 96, 173, 3, 3, 201
OC	1020 data 3, 240, 17, 133, 252, 173, 2, 3, 133
MN	1030 data 251, 169, 99, 141, 2, 3, 169, 3, 141
MG	1040 data 3, 3, 96, 173, 254, 1, 133, 89, 162
DM	1050 data 0, 160, 0, 189, 0, 2, 240, 22, 201
CA	1060 data 32, 240, 15, 133, 91, 200, 152, 41, 3
NG	1070 data 133, 90, 32, 183, 3, 198, 90, 16, 249
OK	1080 data 232, 208, 229, 56, 32, 240, 255, 169, 19
AN	1090 data 32, 210, 255, 169, 18, 32, 210, 255, 165
GH	1100 data 89, 41, 15, 24, 105, 97, 32, 210, 255
JC	1110 data 165, 89, 74, 74, 74, 74, 24, 105, 97
EP	1120 data 32, 210, 255, 169, 146, 32, 210, 255, 24
MH	1130 data 32, 240, 255, 108, 251, 0, 165, 91, 24
BH	1140 data 101, 89, 133, 89, 96

Listing 1b: PET/CBM VERIFIZER (BASIC 2.0 or 4.0)

CI	10 rem* data loader for "verifier 4.0" *
CF	15 rem pet version
LI	20 cs=0
HC	30 for i=634 to 754:read a:poke i,a
DH	40 cs=cs+a:next i
GK	50:
OG	60 if cs<>15580 then print "***** data error *****":end
JO	70 rem sys 634
AF	80 end
IN	100:
ON	1000 data 76, 138, 2, 120, 173, 163, 2, 133, 144
IB	1010 data 173, 164, 2, 133, 145, 88, 96, 120, 165
CK	1020 data 145, 201, 2, 240, 16, 141, 164, 2, 165
EB	1030 data 144, 141, 163, 2, 169, 165, 133, 144, 169
HE	1040 data 2, 133, 145, 88, 96, 85, 228, 165, 217
OI	1050 data 201, 13, 208, 62, 165, 167, 208, 58, 173
JB	1060 data 254, 1, 133, 251, 162, 0, 134, 253, 189
PA	1070 data 0, 2, 168, 201, 32, 240, 15, 230, 253
HE	1080 data 165, 253, 41, 3, 133, 254, 32, 236, 2
EL	1090 data 198, 254, 16, 249, 232, 152, 208, 229, 165
LA	1100 data 251, 41, 15, 24, 105, 193, 141, 0, 128
KI	1110 data 165, 251, 74, 74, 74, 74, 24, 105, 193
EB	1120 data 141, 1, 128, 108, 163, 2, 152, 24, 101
DM	1130 data 251, 133, 251, 96

Bits and Pieces

Got an interesting programming tip, short routine, or an unknown bit of Commodore trivia? Send it in – if we use it in the Bits & Pieces column, we'll credit you in the column and send you a free one-year's subscription to *The Transactor*

1541 Error Allocator Scott Gray, New Bloomfield, MO

Do you have a disk you can't use because of errors on it? Rejoice, for help is here! Type in and SAVE Error Allocator. Place the faulty disk in the drive and RUN the Allocator. In a few minutes every sector with an error on it will be allocated. Now, since DOS won't try to write to those sectors, you can SAVE programs to that disk without encountering the dreaded Read Error!

```

DA 10 rem 1541 error allocator
KI 20 c$ = chr$(147): h$ = chr$(19): l$ = chr$(157):
MN 30 print c$: open 15,8,15,"u;": open 2,8,2,"#"
GD 40 gosub 110: if ef then stop
CO 50 for t = 1 to 35: for s = 0
    to 20 + 2*(t>17) + (t>24) + (t>30)
FA 60 print h$ " track "t" sector "s;
ML 70 print tl$ " [3 spaces] ": print#15,"u1:" 2;0;t;s
    : gosub 110: if ef = 0 then 100
HA 80 print#15,"b-f:" 0;t;s: print#15,"b-a:" 0;t;s
OL 90 print: print "error "er" on "t","s"
PB 100 next: next: close 2: close 15: end
FK 110 input#15,er,er$,e1,e2
    : ef = 1 + (er = 0 or er = 65 or er = 73): return
    
```

Coloured Remarks Without REM

**Luis Pistoia
Argentina**

Here is a simple way to make important REMs, such as those identifying subroutines, to stand out in program listings.

Instead of the usual REM, like

```
1000 rem *** sound routine ***
```

Make your subroutine-identifying remark like this:

```
1000 rem "MQ[YEL]*** sound routine ***Z
```

To get the Reverse-M, enter the line with a space in its place at first, then move the cursor over the space and press RVS ON, shift-M, and RETURN.

When you list your program, the above line will appear in yellow as:

*** sound routine ***

You can use your favorite colour instead of yellow, or use RVS-ON to highlight the message.

Directory Filename Highlighter

**Dino Bavaro
Don Mills, ON**

Here is a handy disk utility which allows you to highlight any filename in the directory. This is useful in making certain programs stand out, such as program boots; or to title various sections of the directory. The highlighting is achieved by renaming the file with four special characters preceding it. The four characters are: shifted space, two delete characters and a reverse-on. This leaves enough space for only twelve characters for the rest of the filename. The routine below will first ask you whether you want to highlight or un-highlight and then ask for the filename. To load a highlighted file from the directory you can use:

```
load " [shifted space]???filename" ,8,1
```

```

NM 10 rem filename highlighter
BN 20 open 15,8,15: gosub 1000
GI 30 hd$ = chr$(160) + chr$(20) + chr$(20) + chr$(18)
FN 40 input "1:highlight, 2:un-highlight ";n
FG 50 on n gosub 100,200
MD 60 end
KL 70 :
BB 100 input "filename to highlight ";hp$
GM 110 print#15,"r0:" + hd$ + hp$ + " = " + hp$
NA 120 gosub 1000 :rem check disk error
OJ 130 return
AA 140 :
MF 200 input "filename to un-highlight ";dp$
KA 210 print#15,"r0:" + dp$ + " = " + hd$ + dp$
BH 220 gosub 1000 :rem check disk error
CA 230 return
EG 240 :
AC 1000 input#15,e,e$,t,s
GE 1010 if e then print e,e$: end
IB 1020 return
    
```


Easy Speedup For The C-128 With 1541 Drive

Richard Young
Greenwood, NS

Here's the BASIC Loader:

Remember the VIC-20 and the 1541 disk drive? Remember the instruction in the disk drive manual for adjusting the 1541 for use with the VIC-20? Actually, the command OPEN 15,8,15,"UI-" was not required to make the 1541 VIC-20 compatible, but rather it allowed the 1541 to function *faster* with the VIC. Now note the FAST command in the Commodore 128 (in 128 mode); it turns off the 40-column screen - no big deal because every 128 user should run it in 80 columns for anything but graphics - and allows the 128 to function much faster. Naturally, it can easily keep up with the 1541 set for the VIC-20 speed. An estimated 17 to 18 percent increase in speed for all disk I/O is the result. With a 1541 and Commodore 128 in 80 columns, enter:

fast: open 15,8,15,"ui-"

Now you're in for faster business without special speed-up software! Note that "UI+" puts the 1541 back to the slower speed required for the 64 mode and 128 in 40 columns. Of course, a 1571 is much faster still. . .

```
EH 10 rem "load.it.here" by frank colaricci
ME 15 rem loads a prg at a given address
IE 20 rem 100% relocatable - edit line 35
HO 25 rem syntax:
IC 30 rem sys a,"filename",device number,
    load address
NI 35 a=49152
LC 40 for i=a to a+72: read b: poke i,b: next
NC 45 end
BG 50 data 032, 253, 174, 032, 158, 173, 032, 143
DF 55 data 173, 169, 100, 160, 101, 032, 219, 182
KD 60 data 160, 002, 177, 100, 153, 251, 000, 136
LI 65 data 016, 248, 165, 251, 166, 252, 164, 253
II 70 data 032, 189, 255, 032, 253, 174, 032, 158
AJ 75 data 173, 032, 247, 183, 152, 170, 169, 008
NH 80 data 160, 000, 032, 186, 255, 032, 253, 174
KJ 85 data 032, 158, 173, 032, 247, 183, 072, 152
PG 90 data 170, 104, 168, 169, 000, 032, 213, 255
PA 95 data 096
```

C-128 Key Repeat

Mort Adler, Winnipeg, MB

The C-128 has auto-repeating keys as a default after power-up. To disable the auto-repeat, simply type:

poke 2594,64.

To re-enable auto-repeat, type:

poke 2594,128

C-64 Load It My Place Not Yours!

Frank Colaricci
Winter Park, FL

The Volume 6 Issue 05 Bits & Pieces section contained a program named "relocate". I would like to suggest a relocatable load as an alternative to editing the load address of a disk file.

Here's a routine that can be appended to your program that will load in PRG files where you want them. The ML program that "LOAD.IT.HERE" creates is relocatable and may be loaded wherever you have 73 bytes of free memory. Please note that locations 251 through 253 are used during the execution of this ML program.

The syntax of using LOAD.IT.HERE is:

SYS start,"filename",device number,load address

C-64 Italics

Glen Mackinnon, Hanover, ON

Do you wish the Commodore 64 had more than one built-in character font? Thanks to the VIC II chip you can design your own custom fonts, unlike PET owners who are stuck with the Commodore set.

Custom character sets are usually reproduced in magazines via hundreds of DATA statements, but this short program creates the new characters by modifying the old character patterns in memory. This program uses the built-in character set to create a new set of pseudo-italic characters.

The program shifts the upper four rows of each character to the right (the N/2 in line 70), giving them an italics-type slant. The new character set is located at 12288 (hex \$3000) and is enabled by a POKE 53272,29. Since this program uses straightforward POKES and only a divide-by-2 (shift right in machine language), it is a challenging but not impossible task for beginners learning machine language to try to translate. In fact, this program should be translated to machine language for increased speed; it is presented here in BASIC to clearly show the method of italicizing the characters.

After the program runs, the upper and lower-case characters are unchanged, but replacing the reverse characters are the italic-like letters (this has the side effect of destroying the cursor). To use the italics, just print or type your desired text in reverse-field. To switch back to the regular character set, use POKE 53272,21.

EI	10 rem italics for the c64
BF	20 poke 53272,29: rem change char. set
PO	30 ad = 55296: t1 = 12288: t2 = 13312
MK	40 poke 56333,127: poke 1,51: rem disable irq, get char. rom
JO	50 for i=0 to 1023: n = peek(ad + i)
CC	60 poke t1 + i,n
AM	70 if (i and 4)=0 then n = n/2
DF	80 poke t2 + i,n: next i
LM	90 poke 1,55: poke 56333,129: re-enable irq, normal rom
HB	100 print "normal characters"
HM	110 print "r italics characters!"

The SCNKEY Kernal Routine (Author Unknown)

Here it is, the naked truth: the SCNKEY Kernal routine has been sadly neglected. Well, no more! For speedy keyboard input, it is unparalleled, a hare to GETIN's tortoise.

To tell you the truth, I'm not even sure how it was that I stumbled across this handy routine (located at 65439 - \$ff9f). I noticed that in all the memory maps, it was marked as not returning any values, but decided to try it out anyway. Voilà! When the routine is called, the Ascii value of the current key down is returned in the .X register!. This was something I'd been looking for for a long time, and has since proved to be superior to GETIN to check for keypresses.

It works this way: the GETIN routine takes a character from the keyboard buffer and returns it in the accumulator. All keys can be made to repeat by putting a 128 in location 650, but a delay loop in the keyboard scan means that only about 7 characters a second are returned.

But SCNKEY has no such limitations. It is called as part of the standard interrupt process and updates the keyboard buffer and all key locations. However, reading from the buffer is slow and the locations (such as location 203) return values other than Ascii codes. It's the .X register that's the key. The value left here takes into account the shift, CTRL and Commodore-logo keys and reflects the state of the keyboard when SCNKEY is called. Best of all, SCNKEY can be called and the keyboard read even when interrupts are disabled. Try this; the speed is amazing.

The SCNKEY routine can be used to great advantage in both ML and BASIC. To use it from BASIC, SYS 65439 and then PEEK(781) to get the Ascii value of the key currently held down. I only hope that future memory maps will give this great little piece of code its due.

C-64 and VIC Un-NEWS Shea T. Small, Thornhill, ON

It's probably happened to everyone. While working on some BASIC program you accidentally type "NEW". Usually that's the end of that. All those hours of work gone down the drain. But there is a way out of that. Enter this line:

```
For the 64 -   poke2050,1:sys42291:poke45,peek(34)
               :poke46,peek(35):clr
For the VIC -   poke4098,1:sys50483:poke45,peek(34)
               :poke46,peek(35):clr
```

Fast Memory Clear Using Garbage Donald Fulton Stoneman, MA

Garbage can be useful. The BASIC line below will clear most of free memory, from 40K to 8K in only 2 seconds. Doing the same job with POKE would take almost 2 minutes.

```
z$ = " ": for x = 1 to 255: z$ = z$ + chr$(0): next
```

In generating one active string of 255 characters, an amazing 32K of dead strings are left behind in dynamic string space. The math is the sum of 1 + 2 + 3 + . . . + 254 + 255.

This technique is effective in clearing a hi-res screen (or filling it with any given byte) if the screen is located within the normal free RAM area (below \$A000). Do the clear before moving the top of BASIC down.

(PRINT AT Update) Update Mike Schmidt North Tonawanda, NY

In the Volume 6, Issue 5 Bits and Pieces column an article titled "PRINT AT Update" stated that the Kernal PLOT routine was unreliable when entered through the jump table at 65520 (\$FFF0).

This is not the case if the carry is set or cleared before calling the routine by POKEing the desired status into memory location 783 (\$030F). Before executing the routine, BASIC will put the contents of 783 into the processor status register.

So, to set the cursor position:

```
poke 781,row: poke 782,col: poke 783,0
: sys 65520: print "message"
```

To read the cursor position:

```
poke 783,1: sys 65520: row = peek(781)
: col = peek(782)
```


User Friendly Commands

Frank E. DiGioia
Athens, GA

Commodore Trivia Department: Did you know that there are two commands on the C64 which cannot give a syntax error no matter what kind of arguments you use with them (if any)? Can you guess which ones they are? (Hint: they aren't STOP and END.)

The Answer: GOTO and GOSUB

Place the following program in memory (so we'll have something to GOTO) and then try to crash the GOTO or GOSUB statements.

```
0 print " this is line 0 "  
1 print " this is line 1 "
```

Now give them your worst:

```
goto (fred)  
goto $$$  
gosub " string var "  
gosub for/next
```

You simply *cannot* crash these commands.

REM RAM: Tag-Along Program Variables

Herbert R. Coburn
Spokane, WA

Here's a trick I have not seen in any magazine or book about Commodore.

The two bytes at \$43 and \$44 point to the start of BASIC text. Start your program with a REM statement and a chr\$(34) to gain a block of reserved memory imbedded in the program. Add six to the pointer at \$43 and it points to the first byte of the reserved block. The block of memory can be as much as 74 bytes, depending on whether or not you allow the line to be listed. The chr\$(34) following the REM token lets most values POKEd into the line to be displayed.

The value of this trick is that the block of memory rides along with the program when it is SAVED. For those of us not enamoured by copy protection, it lets the user copy an installed program without having to worry about tag-along files. I use it to store initial values that depend upon a user's configuration. Simple instructions to the user can guide him, or her, through placing the correct values in the line by using the Screen Editor and SAVEing the program under another name. Or, one can be more ambitious and have the program determine if it has been 'installed'. It can then ask the appropriate questions, POKE the right values, SAVE and RUN itself. This way, all previously set up parameters are there as soon as the program is LOADED - no need to store them in separate files anywhere.

Editor's Note: A good example of this technique is the program which Mr. Cohen enclosed with this article, but was not printed here for lack of space. When first run, the program allows you to set up the background, border and character colours to your liking, then stores these values in the REM statement as described above, and finally, SAVES itself. The next time you LOAD and RUN the program, it restores these colours by looking at the three bytes it stored in the REM statement. Using the REM storage technique, a program can know whether it was run before, and it can find out some information from the previous run as well. -T.Ed

Stringings

Jonathan Hill, Bloomfield, CT

In Commodore BASIC, there are some conditions that occur when using such character string functions as LEFT\$, RIGHT\$, and MID\$ that programmers should be aware of. The condition can first be described when using the function:

```
b$ = left$(a$,3)
```

This command appears fine: B\$ should contain the first three characters that are in A\$. But look again! If A\$ is shorter than three characters, B\$ will also be less than three characters long. This is definitely something to be wary of, as sometimes in a program, strings must be of a known fixed length.

One solution is to use a decision statement before the string function to see if the character string is long enough to fill the bounds set by the string function, but there is a better solution. In the above example if three spaces are added to A\$, then B\$ is guaranteed to be three characters long, even if A\$ hasn't been assigned anything yet and contains a null (a zero-length string). For example:

```
b$ = left$(a$ + " [3 spaces] ")
```

A similar method will work for RIGHT\$ and MID\$, simply adding spaces to the right or left of the character string, where needed. Keep in mind, too, that you can pad the string with characters other than spaces.

Finally, this solution also provides a handy formatting tool for the PRINT statement. C-64 BASIC has no PRINT USING command, such as is used on other computers to set up fixed length output fields, but you can achieve the same effect by printing strings of a fixed length using the above technique. Numerical data can be formatted by first converting to a string with the STR\$ function. The following are examples:

```
print right$(" [4 spaces] " + c$,4)  
print " amount payable ";right$(" ***** " + str$(amt),6)  
print mid$(e$ + " [2 spaces] ",2,1)
```


Rhetorical Loops

The following program is just a nine-digit counter which starts at zero and counts up. Big deal. But it uses nested FOR/NEXT loops to do it, and the variable names used are unconventional enough to turn the code into readable nonsense. Here's the program:

```
1 rem "Programmable Prose by Chris Zamara
  and Nick Sullivan
2 mo = 9:ol = 9:in = 9:om = 9:od = 9:ps = 9:ro = 9:r = 9
3 :
100 ford = automobile
110 fork = food tool
120 for a nice time = call antoinette
130 forty thieves = far too many
140 forsaken = stood up
150 forest = treetops
160 fort knox = hard to rob
170 formula = highly top secret
180 foreigner = visitor
190 :
300 print d;k;an;ty;sa;es;tk;mu;ei
310 next ei,mu,tk,es,sa,ty,an,k,d
```

The above program is fairly useless, but shows that if you really want to, BASIC will let you write strange-looking and difficult to understand code. It's the programmer's responsibility to use meaningful variable names and put spaces in the right place. On the other hand, perhaps "programmable prose" could be a new form of expression.

SYSing With The C-128

The C-128, like the 64, allows you to pass values to a machine language routine through the A, X and Y registers, and also read the contents of these registers after the routine has finished. You can POKE the desired values into special RAM locations to initialize the registers, SYS to the routine, then PEEK the locations to find the values set by the ML routine. The locations are as follows:

Location		Register
C64	C128	
780	6	Accumulator
781	7	.X register
782	8	.Y register
783	5	.P (Processor Status) register

But wait! With the C-128 it's even easier than that. You can pass values of A, X, Y and P to a machine language program *directly from the SYS statement*. To do this, just put the parameters after the SYS like this:

SYS address,A,X,Y,P

Any or all of the parameters can be left out, and they will default to what's in the memory locations indicated above. For example, you could code "SYS addr,20" to set the accumulator to 20, or "SYS addr,,,1" to set the carry flag (processor status = 1) before entering the routine.

Using your favorite Kernal routine from BASIC has never been simpler than on the 128! You can make your BASIC programs more efficient by making calls directly to ROM routines. This makes for awful, unportable code, but if you need more speed, a few strategic SYSes may do the trick.

Running ABasiC From The CLI On The Amiga

**Robert Case
Springfield, OR**

To run ABasiC from the CLI (instead of by clicking the ABasiC icon from WorkBench), you have to first increase the stack size to prevent a crash. From the CLI, enter:

```
STACK 8000
RUN ABASIC
```

ABasiC will then come up on a screen of its own. To return to the CLI or WorkBench, first reduce the ABasiC window size and move the window to reveal the ABasiC template on the top line of the screen. You can then slide down the ABasiC screen or click it behind with the re-ordering gadgets to reveal the WorkBench screen and other open windows.

C-64 Auto-Start

**Steen Pederson
Frederiks, Denmark**

Here is a compact BASIC program which will turn any program into one which will automatically RUN when loaded. Just enter and run "Autostart 1.1" listed below, and it will ask for the name of the program to convert. It then asks for the name of the new, auto-start version of the program to put on disk. After a while (depending on the length of the program), Autostart will finish, but leave the machine in a confused state. You'll have to reset the machine (turn OFF then ON) at this point.

The new file created on disk must be loaded in the following way:

```
load "filename",8,1
```

Your program will then LOAD and automatically RUN. The STOP and RESTORE keys will be disabled, so your auto-start programs will be protected from being modified and re-saved to disk.


```

BE 100 print "Enter name of program to be auto-started "
CK 110 input "present name ";f$
AO 120 print "Enter filename for new auto-boot program "
KM 130 input "new name ";n$
MH 140 poke 649,0: open 1,8,1,"0:" + n$
CO 150 print#1,chr$(199);chr$(2);
EC 160 for i = 1 to 61
FL 170 read v: print#1,chr$(v);: next
CD 180 for i = 772 to 2048
OG 190 print#1,chr$(peek(i));: next
NO 200 open 2,8,3,f$: get#2,g$,g$
CI 210 for i = 0 to 1
AC 220 get#2,g$: if g$ = "" then g$ = chr$(0)
OH 230 i = st: print#1,g$;: next
PK 240 close 1: close 2
OG 250 :
PE 260 data 169, 47, 133, 0, 169, 55, 133, 1
GD 270 data 169, 0, 133, 157, 32, 68, 229, 169
MJ 280 data 82, 141, 119, 2, 169, 213, 141, 120
JK 290 data 2, 169, 13, 141, 121, 2, 169, 3
IG 300 data 133, 198, 169, 131, 141, 2, 3, 169
EG 310 data 164, 141, 3, 3, 169, 52, 141, 20
GF 320 data 3, 169, 173, 141, 24, 3, 76, 116
CM 330 data 164, 139, 227, 199, 2

```

The C-64 Great Escape

David Claussen
Menomonee Falls, WI

Below is a short machine language program for the C-64 that creates the effect of an escape key. An escape key, which is normally found on IBM compatibles, is normally not available for Commodore owners. What an escape key does, simply put, is let the user "escape" from whatever he or she may be doing at the time.

While programming in BASIC, what this function does is clear the screen line that the cursor is on and position the cursor at the first column. It also turns off quote mode, which allows normal use of the cursor and other control keys. And finally, it turns off reverse mode.

The back-arrow key (located at the top left corner of the keyboard) becomes the escape key. If you wish to use the back-arrow key in a program, hold down the Commodore-logo key while pressing back-arrow.

```

FB 10 rem ** escape key - press backarrow **
FM 20 for j = 49152 to 49227: read x: poke j,x
    : ck = ck + x: next
HF 30 if ck <> 8698 then print "data error": stop
FB 40 sys 49152
GK 50 :
GJ 100 data 120, 169, 13, 141, 20, 3, 169, 192
AM 110 data 141, 21, 3, 88, 96, 165, 197, 201

```

```

OJ 120 data 57, 208, 54, 173, 141, 2, 201, 2
KL 130 data 240, 47, 166, 214, 32, 255, 233, 160
OM 140 data 0, 24, 32, 10, 229, 169, 29, 141
BJ 150 data 119, 2, 169, 157, 141, 120, 2, 169
JL 160 data 2, 133, 198, 169, 0, 133, 212, 169
CH 170 data 0, 133, 199, 169, 32, 141, 119, 2
FN 180 data 169, 157, 141, 120, 2, 169, 2, 133
LG 190 data 198, 76, 49, 234

```

Return of The Swords Of Doom

Arthur Wolf
Wichita, KS

We told you they'd be back! This time, thanks to Mr. Wolf's program, the Evil Swords appear as comets! Ooh, scary stuff, kids!

```

EP 10 rem for frustrated comet gazers
JE 15 rem here's this rendition of chris's
KN 20 rem "Evil Swords of Doom" (6/4 p.9)
NI 25 :
BP 30 l$ = chr$(157)
KG 35 poke 53280,0: poke 53281,0
AB 40 a$ = "[BLUE]M[q]M[q][CYAN]M[q]M[q][WHT]
    [Q]Q]Q]Q]Q] " + l$ + l$ + l$ + l$ + l$ + l$ + "[q]"
FP 45 b$ = "[q] [q] [q] [q] "
MC 50 print chr$(142)
OC 55 print "s" tab(rnd(1)*41)
LM 60 for i = 1 to 19: print a$;
FP 65 for d = 1 to 15: next
BE 70 next i: print b$;: goto 55

```

Date Conventions

R.C. Eldridge
Pemberton, BC

There has always been confusion because of the U.S. convention of expressing the date in the order "month-day-year" and the rest of the world using day-month-year. Several years ago an international standards body recommended that everyone use year-month-day and it is slowly catching on.

If you use year-month-day as a date reference in one field of a data file, the computer will automatically sort references into the right chronological order. For logging purposes where precise time is important the number can be extended to year-month-day-hour-minute.

The basic idea is useful in a two-day amateur radio contest log. If you use the form day-hour-minute for the time entry, the log can be re-sorted easily into chronological order after having sorted into callsigns or countries or whatever for analysis.

The Hidden Message

Jim Butterfield, Toronto

There's an encrypted message in the Commodore 128. You'll never find it by inspecting memory, since it is definitely in code . . . and not an easy one to crack.

I'll tell you where it is located. It's in bank 15 – that's ROM – at addresses 44644 to 44799. It's not easy to crack; since every one of the 156 characters has a different "key" value, it's not a simple Caesar cipher. In fact, if the 156 keys were independent and random, the code would indeed be uncrackable, since no key is repeated. But each key is mathematically related to the previous one, and a cracker with time and ingenuity might – perhaps – be able to break it. I'm about to give it away, so you might like to stop reading right now if you're a serious cipher solver.

Secret code can often be found in software. Sometimes it's a personal signature by the author. Sometimes it's a secret proof of copyright. Sometimes it's an amusement. Here's a simple Basic program to make the code readable.

By the way, if you just want to read the message, I'll give you a quick method at the end of this article.

Enter the following crude decoding program on your Commodore 128. Use 40 column mode, because I'm POKE-ing to the screen.

```
100 bank 15
110 print chr$(147);chr$(14)
120 print:print:print
130 for j = 1 to 156
140 x = xor(xor(peek(44643 + j),j),59)
150 m = 192:if(x and m) = 0 or (x and m) = m then m = 0
160 if (xor(x,m) and 32)>0 then m = xor(128,m)
170 poke 1023 + j,255 and xor(j,m)
180 next j
```

You'll see the message in crude screen format – formatting characters such as RETURN will appear as control characters, but it's readable.

If you just want to read the message, and don't care where it's stored or in the decoding process, there's an easier (and neater) way to see it on your 128 screen. Just type:

sys 32800,123,45,6

Verfizer For The Plus 4 and C128

By next issue we'll have a Verifier for the B Machines and for the C128 in 80 Column mode. They'll all appear up at the front with the other Verifier programs.

Plus 4 Verifier

```
NI 1000 rem * data loader for "verifier +4"
```

```
PM 1010 rem * commodore plus/4 version
EE 1020 graphic 1: scnc1r: graphic 0: rem make
    room for code
NH 1030 cs = 0
JI 1040 for j = 4096 to 4216: read x: poke j,x
    : ch = ch + x: next
AP 1050 if ch<>13146 then print "checksum error"
    : stop
NP 1060 print "sys 4096: rem to enable"
JC 1070 print "sys 4099: rem to disable"
ID 1080 end
PL 1090 data 76, 14, 16, 165, 211, 141, 2, 3
CA 1100 data 165, 212, 141, 3, 3, 96, 173, 3
OD 1110 data 3, 201, 16, 240, 17, 133, 212, 173
LP 1120 data 2, 3, 133, 211, 169, 39, 141, 2
EK 1130 data 3, 169, 16, 141, 3, 3, 96, 165
DI 1140 data 20, 133, 208, 162, 0, 160, 0, 189
LK 1150 data 0, 2, 201, 48, 144, 7, 201, 58
GJ 1160 data 176, 3, 232, 208, 242, 189, 0, 2
DN 1170 data 240, 22, 201, 32, 240, 15, 133, 210
GJ 1180 data 200, 152, 41, 3, 133, 209, 32, 113
CB 1190 data 16, 198, 209, 16, 249, 232, 208, 229
CB 1200 data 165, 208, 41, 15, 24, 105, 193, 141
PE 1210 data 0, 12, 165, 208, 74, 74, 74, 74
DO 1220 data 24, 105, 193, 141, 1, 12, 108, 211
BA 1230 data 0, 165, 210, 24, 101, 208, 133, 208
BG 1240 data 96
```

C128 Verfizer (40 column mode)

```
PK 1000 rem * data loader for "verifier c128"
AK 1010 rem * commodore c128 version
JK 1020 rem * use in 40 column mode only!
NH 1030 cs = 0
OG 1040 for j = 3072 to 3214: read x: poke j,x
    : ch = ch + x: next
JP 1050 if ch<>17860 then print "checksum error"
    : stop
MP 1060 print "sys 3072,1: rem to enable"
AG 1070 print "sys 3072,0: rem to disable"
ID 1080 end
GF 1090 data 208, 11, 165, 253, 141, 2, 3, 165
MG 1100 data 254, 141, 3, 3, 96, 173, 3, 3
HE 1110 data 201, 12, 240, 17, 133, 254, 173, 2
LM 1120 data 3, 133, 253, 169, 38, 141, 2, 3
JA 1130 data 169, 12, 141, 3, 3, 96, 165, 22
EI 1140 data 133, 250, 162, 0, 160, 0, 189, 0
KJ 1150 data 2, 201, 48, 144, 7, 201, 58, 176
DH 1160 data 3, 232, 208, 242, 189, 0, 2, 240
JM 1170 data 22, 201, 32, 240, 15, 133, 252, 200
KG 1180 data 152, 41, 3, 133, 251, 32, 135, 12
EF 1190 data 198, 251, 16, 249, 232, 208, 229, 56
CG 1200 data 32, 240, 255, 169, 19, 32, 210, 255
EC 1210 data 169, 18, 32, 210, 255, 165, 250, 41
AC 1220 data 15, 24, 105, 193, 32, 210, 255, 165
JA 1230 data 250, 74, 74, 74, 74, 24, 105, 193
CC 1240 data 32, 210, 255, 169, 146, 32, 210, 255
BO 1250 data 24, 32, 240, 255, 108, 253, 0, 165
PD 1260 data 252, 24, 101, 250, 133, 250, 96
```


C-128's Help Key Redefined

Walter Kiceleff
Buenos Aires, Argentina

I really like your magazine and I would like to contribute by sending you this 'Curiosity' I discovered in my computer.

In the Commodore 128, you can redefine the 'HELP' key to use it like a Function key. The HELP key has a memory assignment of only 5 bytes (4168-4172). If you Poke these locations with the Ascii value of the characters you want to use; Presto! It works. For example:

```
10 for i = 4168 to 4172
20 read a$: poke i,asc(a$)
30 next i
40 data p,r,i,n,t
```

If you want to add a carriage return, poke 4172,13. Then your message will have 4 letters (not 5) plus a carriage return.

Amiga Lattice C Notes

Robert Case
Springfield, Oregon

While using the Amiga Lattice C Compile, version 3.02, I encountered two problems. A description of each problem follows:

First Problem: *scanf*. . . When *scanf* was used in the following form, the program didn't halt and wait for keyboard input:

```
scanf ( " %f%c\n" , &first, &second);
```

When the form was changed to:

```
scanf ( " %f %c\n" , &first, &second)
```

The program would halt and wait for keyboard input. The addition of a space before each '%' corrected the problem.

Second Problem: letters "E" and/or "e" . . . when these letters were used as a key to "exit" the program using 'scanf', their use was not recognized. When the letter "Q" was substituted for the letter "E", the program worked as expected. Is it possible that the letter "E" is a reserved word in this version of C?

Another point: it is often helpful while working from the CLI to have a larger STACK to prevent a system crash. I reset the stack to 8000 or even higher before programming and testing. (*See next Bit*)

Reading 8250-Formatted Disks with an 8050

Since the 8250 uses both sides of a diskette, you can't use an 8050 to read any data on the opposite side. Fortunately, the 8250 only uses the other side when the first side gets full. To find out if any data has been placed on the opposite side, check the number of blocks free. If there's less than 2052 blocks free, you can read all the files on the disk with an 8050.

1571's Can Be too Smart

We all know that the 1571 is the greatest thing since the return of the mini-skirt. But it tries so hard that it can confuse instead of help. For example, consider this sequence of events:

- 1) LOAD in a short (1 block) program from the C128
- 2) Remove the disk, put it into a 1541 and use a C-64 to replace the program with a different one
- 3) Put the disk back into the 1571 and LOAD "*" ,8

You would expect it to load in the new version of the program, right? Well, the 1571 still has the program in its RAM buffer, and thinks it can be smart and save time by giving you the copy from RAM instead of going out to disk. So, you get the original, un-modified program that doesn't even exist on the disk anymore. Proof that "A little knowledge is a dangerous thing". Or, as Nietzsche put it, "Better know nothing than half-know many things".

Holy Input-Buffer, Batman!

"Robin. . . I think there is a diabolical plot brewing at Commodore headquarters."

"Say it isn't so, Batman!"

"I'm afraid it is. Are you familiar with the Input buffer in the Commodore 64?"

"Well, Batman, from the Bat-Computer I've learned that BASIC program lines are stored in the input buffer after they're entered."

"Right Robin! I believe that after the lines are stored, they are also tokenized in the buffer, are they not?"

"Gosh, Batman, you're so right!"

"Very perceptive of you, Boy Wonder. Now I have but one more question for your keen mind: is the line number stored in the Input Buffer?"

"This one I know! Just like the Bat-Buffer here in the Cave, the C-64 Input Buffer only stores the line itself, without the line number."

"Excellent Robin! Now look at this Bat-Dump of the Commodore 128 Input buffer right after a line has been entered."

"Holy RAM-Chip! There's the line number at the start of the buffer, just as it was entered by the user, leading zeros, trailing spaces and all! In ASCII! What kind of a fiend would concoct such a scheme?"

"The world is full of forces we don't understand, Robin. Commodore is just one of them. They've done the same with the Plus 4 and C-16 as well! Just thank the good Lord that we have Bat-Dumps, Bat-Anthologies, and the minds to comprehend them."

"Well said, Bat-Friend!"

Letters

Help: Line Scanner Required: Some time ago, in 'some' computer magazine, I saw an advertisement for what I believe might have been called a 'line copier' or 'line scanner' for PCs. It was supposed to be able to copy printed or typed text directly from paper and put it in the computer's memory and, presumably, onto disk as sequential files. Is such a device available for the Commodore 64?

William R. Carr
R.R.#3, Box 233
Harrisburg, IL, USA
62946

About a year ago, The Transactor appeared at a computer show called the Computer Fair in the heart of Toronto. The booth next to ours was displaying a product that really caught our eye. It was a line scanner called the Omni-Reader. It worked in a novel manner: the "eye" of the reader was mounted on a vertical/horizontal slider assembly. A document would be placed underneath it and you would scan each line by hand with the slider. At a pretty good pace, it would recognize about 6 or 8 different type fonts, and send out their character codes to an RS232 port. We were impressed. So, in the true spirit of advancing with the times, Karl struck up a conversation with the sales rep, who promptly agreed to lend us one for a few days to try out.

Well, the people representing the Omni-Reader must have fallen off of the edge of the world because we never heard of them again. Too bad; an item like that deserves some terrific free press. My advice today is to hope that this letter/reply will generate some response from our reading audience. If anyone reading knows of a line scanner for the Commodore 64, or for that matter any computer, please drop us or William a line. We would really appreciate it.

Attention Hot 1541 Owners: Here's a bit of helpful information to help hot disk drive users to cool down.

An easy and inexpensive way to prevent overheating of your 1541 disk drive is to buy four new pencils and cut them down to two and a quarter inches, measured from the end of the eraser. Bevel the cut ends slightly with a pencil sharpener. Place the bevelled ends into the recessed screw holes on the underside of the disk drive. This allows fresh air to get to the breathers located on the underside of the drive.

I would also like to ask a question. If I get a C-128 will my C-64 modem be compatible with the C-128 in all modes? What about an interface for a printer?

Duane Barry, Cambridge, Ontario

A 1541 on four legs. It might catch on. I am pretty sure that 'hot' 1541 users all over will appreciate your advice. Thanks.

About the C64 modem. The modem will work in the C128 mode but not the CP/M mode, at least not yet. The modem itself is not at fault on this one. It's just that Commodore did not include a driver for the RS-232 port just yet in their release version of CP/M Plus. Word's out, though, that a version is available through CompuServe that supports RS-232 communications, and that within a few months Commodore will be releasing the same for general public consumption. About CompuServe and CP/M: apparently, Commodore has included some extra software that re-configures your C128 in C128 mode to act like a CP/M machine so you can download the new CP/M and store it on a CP/M formatted diskette. Just make sure that you are in C128 mode with an appropriate terminal package when you phone in, and have a CP/M formatted disk handy.

A regular Commodore serial printer will work in all modes of the C-128. If you have an interface that hangs off the serial port to some strange type of printer, chances are that it will work just fine. But if your interface, whatever the type, connects to the cartridge port, as with an IEEE interface, you can be pretty well assured that the C128 won't like it.

Take for example an IEEE interface that we all use at the magazine. It has been dubbed the GLINK (Garvin's Link). It is a terrific true-to-life IEEE interface for the Commodore 64. Its true beauty lies in the fact that it doesn't do anything: no extra commands and no special tricks. It just supplies a really fast IEEE interface for your 64 without consuming memory. It does this by swapping itself into 1/2 of the E ROM. The RAM underneath is left alone, assuming that one lead is hooked up correctly inside of the 64 to a resistor. But the C128 is a totally separate system and the GLINK is not compatible with C128 ROM.

However, the GLINK will work on the C128 in 64 mode, as will most of the cartridge port cards for the 64. Be careful though - some have leads that are connected internally and the C128 PC board is much different than the C64. The GLINK, for instance, has a lead to the left lead of R44 on the 64; on the C128 it goes to Pin 29 of the OS8502 chip. This little trick was supplied to us by a gentleman we met recently while out in San Francisco.

One last point: the GLINK works fine with both Viewtron and Quantum Link downloads. The Viewtron software loads and runs fine from the IEEE drives, but the Quantum software must be loaded from a 1541 or compatible. But Quantum downloads to the IEEE work - just flip the GLINK switch back to serial when it's done.

"... but if you fool me twice then I'm indeed a fool.":

Thank you very much for publishing John Holttum's 'The Commodore 128: Impressions and Observations' in Vol. 6, Issue 5. Like many C-64 owners I suspect, I felt wine and dined by Commodore's advance advertising for the new C-128. But I was mildly suspicious because of their poor record with the Plus/4 and C-16, notably their failure to provide good documentation and programmer's support for those models, i.e. something akin to the 'Commodore 64 Programmers Reference Guide'. Also, the C-1541 disk drive's SAVE@ and SCRATCH (yes Ma, there's a SCRATCH bug too) bugs are a perpetual pain in the you-know-what for (a) programmers during the process of writing code, and (b) for the prospects of reliable database systems which involve scratching or replacing files on disk.

Thanks to Mr. Holttum and The Transactor, I no longer have any problem deciding whether to purchase the C-128/1571 system. Actually, I decided that my next disk drive would have to be a dual drive and presumably the 1572 is essentially just two 1571's in the same case. But in any case, I will not be buying the C-128/1571/1572 until there are plenty of independent public reports that Commodore has remedied the above mentioned problems. You can fool me once (the C-1541), but if you fool me twice then I'm indeed a fool.

John R. Menke, Chessoft Ltd., Mt. Vernon, IL

Nice to hear from you again John. There is a rumour that new 1571 ROMs are under construction that fix Save@ as well as other bugs. When it will be released and under what kind of offer we probably won't know 'till it's ready.

Documentation seems to be coming. Commodore is releasing a technical reference through SAM's again, Abacus has the "Internals" book and another on the way, and Jim Butterfield's book will be updated too.

Relative File Access In ML: Loved your excellent article on disk access from machine code in Volume 6 Issue 5, and yes I would like to see more code on the use of relative files.

So here's some stuff. The syntax in Basic for relative files is:

```
OPEN the command channel OPEN 15,8,15
OPEN the relative file OPEN 2,8,2, "0:filename"
Set the POINTER to the record with:
PRINT#15, "P" + CHR$(channel# + 96) + CHR$(lo-rec#)
+ CHR$(hi-rec#) + CHR$(character)
WRITE or READ the file with PRINT#2, INPUT#2, or GET#2
CLOSE the channels after use.
```

One word of caution; if you write to the file, IMMEDIATELY after writing the file, reset the pointer to the beginning of the file accessed with a recall to the set POINTER routine. This will stop any mess-up of files.

Sorry I don't have PAL but I'm saving my pennies up to get it. In machine code the routines are similar to your article.

OPEN Command Channel

```
lda #0f
tay
ldx #08
jsr $ffba ;setlfs
lda #00
jsr $ffbd ;setnam
jsr $ffc0 ;open
jsr $kerr ;your kernal error routine
```

OPEN Relative File:

```
lda #02
tay
ldx #08
jsr $ffba ;setlfs
lda #<nam ;lo address file name
lda #>nam ;hi address file name
jsr $ffbd ;setnam
jsr $ffc0 ;open
jsr $kerr ;kernal error
jsr $derr ;disk error check routine
```

Pointer Routine

```
ldx #0f
jsr $ffc9 ;chkout channel 15
ldy #00 ;length of word to send
;
load = *
lda word,y ;load word
jsr $ffd2 ;output word to command channel
iny
cpy #05 ;end of word yet?
bne load
jsr $ffcc ;clrchn
;
word .byte $50, $5c, $01, $00, $01
```

The characters in WORD are:

\$50 = Ascii for the letter 'p'
 \$5c = Ascii for channel # + 96 (2 + 96)
 \$01 = Ascii for lo byte record # (#1)
 \$00 = Ascii for hi byte record # (#0)
 \$01 = Ascii for character # (first char)

To access any record all you have to do is update the 3rd, 4th, and 5th characters of WORD (lo/hi byte record#) before you call the POINTER routine. To write or read the data use the appropriate input or output routines as in your article.

It's easy and simple to use relative files. They are fun, fast and NOT computer memory robbers. Hope this info is of use.

John Houghton, Collingwood, Ontario

It's nice to hear a kind word mixed with some good advice. Thanks for all. I agree that a pretty large hole was left in my article by excluding relative file access, but I felt it hard to write pure theory without including a ML relative file access demo that worked in a

friendly way, ie. verbose. As you have shown, ML relative file access is not code consuming. At the time I knew that the only code consuming part would be through trying to make it easily useable. Perhaps in a future issue I'll write up a good and friendly data base or something that people can use and learn from. Might be worth a shot.

In case it hasn't been noticed yet, I did encourage a little bit of bad practice in my article with the file read/write technique employed. With the PET, CBM, VIC and 64, code such as:

```
ldx #lfinp
jsr chkin      ;set input device
jsr chrin      ;get a character
pha
ldx #lfout
jsr chkout     ;set output device
pla
jsr chrout     ;write the character
```

Would have been acceptable, as stated in the article. Unfortunately, with a machine such as the C128 a mess would have developed. You have to make sure that CLRCHN was performed before setting either the input or output channels. For the example above, the statement 'JSR CLRCHN' should be inserted before the 'LDX #LF' for both input and output. This is actually good practice irregardless of the machine you are working on. A temporary lapse into bad form caused this unfortunate slip. Sorry about that.

Real Programmers. . . In light of my first letter, I thought I'd better send you this. I found it after many hours of research (i.e. I got lucky looking through some old files). At any rate, I would really like to thank you for giving the ICLIG some free press.

Anyway, I have enclosed my subscription for The Transactor. Nasty trick on your part. Raising the price so we get a better discount. Gee, I wish I would have thought of that sooner.

Kent Tegels
Manager: International Commodore Language Interest Group
18112 North I
Fremont, NE, 68025

Real Programmers Don't Write Specs by Peter S. Hill NCA Corporation

As taken from 'The Special Character Set' - September 1, 1983

Real programmers don't write specs - users should consider themselves lucky to get any programs at all and like what they get.

Real programmers don't comment their code. If it was hard to write, it should be hard to understand.

Real programmers don't write application programs; they program right down on the bare metal. Application programming is for feebs who can't do systems programming.

Real programmers don't eat quiche. In fact real programmers don't know how to SPELL quiche. They live on Twinkies, Doritos, Coke and Swechwan food.

Real programmers don't write in COBOL. COBOL is for wimpy applications programmers.

Real programmers' programs never work right the first time. But if you throw them on the machine they can be patched into working in 'only a few' 30-hour debugging sessions.

Real programmers don't write in FORTRAN. FORTRAN is for pipe stress freaks and crystallography weenies.

Real programmers never work 9 to 5. If any real programmers are around at 9 AM it is because they were up all night.

Real programmers don't write in BASIC. Actually, no programmers write in BASIC after the age of 12.

Real programmers don't write in PL/1. PL/1 is for programmers who can't decide whether to write in COBOL or FORTRAN.

Real programmers don't play tennis, or any other sport that requires you to change clothes. Mountain climbing in OK, and real programmers wear their climbing boots to work in case a mountain should suddenly spring up in the middle of the machine room.

Real programmers don't document. Documentation is for simps who can't read the listings or the object code.

Real programmers don't write in PASCAL or BLISS or ADA or any of those PINKO computer science languages. Strong typing is for people with weak memories.

We do receive the odd piece of mail from time to time. Thanks for relaying that strange bit of tongue-in-cheek programming advice. My addition today to Mr. Hill's list is 'Real programmers do it in their drives!'. A bit wierd but it seems to follow the pattern.

Help Required: I am looking for people interested in helping me type in the New Testament using a word processor. The processor I am presently using is SpeedScript, but another processor would be acceptable as long as the files are compatible, or could be converted for our use. I am using a Commodore 64, with a 1541 disk drive.

After collecting, compiling and editing all the incoming data, I would distribute the finished work to all the participants. If you are interested please call or write for assignments.

Randall J. Bernard
Box 630
Morenci, Arizona
85540 (602) 865-3550

Wow! What a doozy of a task. With a good database and indexing system though, it would be a terrific item. The ability to search the

New Testament via disk would be ideal for report references. Let us know when it's done.

The Drive Disaster: Re: Trans. Disk #10. A Frantic Wave-Off! I tried 'Improved 1541 Head-Cleaning Program'. DISASTER! Drive was 100% OK before using prg. Drive is now in shop for realignment!

D.C. Kerrigan, Greenville, SC

The program as listed in the magazine, Volume 6 Issue 05 page 6, is perfect, as is the copy on disk #10. Although we can assure you of this and feel confident that the code was OK, your drive is still in intensive care. For this we offer two possible explanations.

1) The Commodore drives have an awful habit of getting stuck at times, causing them to no longer function properly for apparently no reason. The real crunch is that even after powering down, the drive doesn't return to normal. Often times this prompts people to bring their drive in for service. Unfortunately, this is often a waste of time and money. A simple initialization of a diskette in the offending drive will cure the problem. This strange occurrence can be traced back to the drive's head being pushed out to an extreme position in either direction. Once in that position there exists a chance that the head will get stuck. Once stuck nothing but a drive initialization or a little internal push on the mechanism will help. Your drive may have been one of the unluckies that gets stuck in extreme positions.

2) The 'Improved Head-Cleaning Program' article stated that the program was not to be run twice in a row, as the quote to follow explains:

"The NEW at the end of the program is not an attempt at program protection, it's there as drive protection. This direct method of stepping the head does not update location \$24. If the program was immediately rerun, the drive head could end up being stepped to track 35 or to bump up against the stop at track 0."

As a test, I deleted line 460 of the program then ran it for the first time. Following that I immediately re-ran the program to see what would happen. Around track 22 the drive mechanism started making an awful noise and continued to do so through track 35. Following this, I loaded in the directory. The drive chattered a bit initially but did finally load the directory. There was no permanent damage to the drive. My drive is almost new and in perfect mechanical shape. A drive that has had a few miles on it might not have fared as well. If this was the case, I still feel that an initialization would force the drive's head back into reality once again. Although running the program twice in a row would have been almost impossible as it was supplied on disk and listed in the magazine, it could have been accomplished as I stated above. Your problem can probably be written off to explanation #1. Just remember, when in a bind, initialize.

Verifizer Update: After recently retiring from 21 years of designing 'Little Black Boxes' for Cessna Aircraft Co., I purchased a C-64.

I am primarily interested in graphics and animation. What little I have learned so far seems to indicate that machine language is the way to go. In pursuit of this I have been attempting to learn what I can about ML but have been disappointed with what I have found. It seems to me that your publication has much to offer towards this goal. . .

. . . One small problem: When using 'Verifizer', the left character of the check signal hides in the upper left corner of the monitor. How do I move it about two spaces to the right?

W.D. Ackerson, Wichita, Kansas

As you have discovered, The Transactor lives for machine language. However, there are a few good books on the market to teach you the basics through extremes of talking to your computer in its mother tongue. One book, which I can't say enough good things about, is Jim Butterfield's Machine Language Book. Published by Bradey/Prentice-Hall, it's an educational dream front to back. If you ever see it in a book store, do it the service of a quick look-over. You will probably be impressed.

About your Verifizer blues: there is a cure. Steve Walley, a reader in Sunnymead CA, ran into the same problem that you did, and as such sent us his modified version of Verifizer that prints two sets of Verifizer checksums on the screen. See 'Double Verifizer' in Volume 6 Issue 06 on page 5.

Sky Travel Support: I enjoyed the review of Sky Travel and would like to mention that I agree with your assessment of the program. What's more. I might just mention a couple of quick utilities:

A seasoned veteran amateur astronomer friend of mine was so delighted with Sky Travel, he dumped his color computer system and purchased a Commodore 64. During December and January we used the program to locate Halley's Comet (as well as several other objects) with surprising accuracy. As a rank amateur astronomer, (I barely know the correct end of a telescope to look in) I was able to locate the comet using hard copy from the program, a compass and binoculars. However, for those with sophisticated systems, the data generated for right ascension and declination seem to be right on the button (provided of course your location and times are correct).

Several friends of mine who are also ham radio operators, are experimenting with using the program for moon bounce. The tracking feature and program's apparent accuracy make this a natural.

I have also used the program with my children (and myself) to become a bit more familiar with the southern New Jersey skies (when the garbage in the air is not too bad).

It's a first rate package and one which ought to cost at least three or four times more than it does. . . an extraordinary buy for \$29.95 - especially given what you can do with it and the information it contains. . . and especially given how much is charged for many poor packages.

Commodore did us a favour putting that one out. . . hope people do take advantage of it.

Peter R. Bent, West Deptford, NJ

Frank Covitz reads The Transactor. Frank Covitz wrote Sky Travel. I am pretty sure that Frank is smiling right about now. But only about the compliment. It seems that Sky Travel is often times pretty difficult to locate. Frank wrote the package but Commodore kind of distributed it. At one point, right about the time that my review was published, Sky Travel was close to being listed as missing in action. But a mixture of public pressure and common logic brought the Sky Travel back from the dead into retail distribution once again. If for any reason anyone would like to get a copy of Sky Travel but can't find it anywhere, then either phone or write Commodore direct, or if that doesn't work, drop us a line. We'll make sure that your request does not go unheard. Frank did too good of a job to allow Sky Travel to fade away so easily.

LADS to PAL Conversion: My recent subscription to The Transactor has gone far in rescuing this amateur from some sort of computer oblivion. Other mags are just fine and often very helpful but games and ads get in the way quite a lot. In The Transactor one finds a balanced, practically fat-free diet of pertinent, challenging and useful information. In short, I'm a very happy customer.

Quite apart from this statement of unbridled joy, I found a statement by Nick Sullivan on page 15 of my first issue (Vol.6, Issue 03) indicating that his Transbasic practically requires the use of a PAL assembler.

I use a RAM-based version of Richard Mansfield's LADS/64 assembler and find it very dependable and easy to use.

Without experience with any other assembler, I find it difficult to decide whether it would be possible to translate Transbasic for assembly with the unit I use. Your advice would be helpful at this point.

R.G. Tischer, Starkville, MS

To best help out everyone trying to convert PAL format to their own special brand of assembler, it might be best if I run down a few of the main PAL directives to be found in The Transactor. They are as follows:

.OPT

This pseudo-op is a directive of output (OutPuT). There are a number of ways to use it. For example:

- .OPT N ;Outputs nothing. Just checks assembly to see if errors exist.
- .OPT OO ;Outputs object to origin (memory).
- .OPT O8 ;Outputs object to device #8 (ie. OPEN 8,8,1,"0:filename" before)
- .OPT P ;Outputs source listing to the default output device during assembly
- .OPT P4 ;Outputs source listing to device #4 (ie. OPEN 4,4 executed before)

Further to this, .OPT can be forced to perform multiple directives of output. For example:

.OPT O8, P4 ; would output the Object to unit 8 and Print the source listing to unit #4.

To continue, you will notice a SYS700 at the start of all PAL source listings for the 64. This calls PAL so that whatever follows will be treated as assembler source code. For other assemblers, this is either omitted or substituted with whatever command starts up the assembly process.

To set the origin that you would like your code to be assembled, you would use a statement such as this:

*** = \$C000**

The '*' represents the current program counter, so in effect you are telling the assembler that the program counter should equal \$C000. Some assemblers use .ORG but the "splat" is more common.

.WORD and .BYTE

These pseudo-ops allow either bytes to be assigned to RAM, or space to be set aside for the same. Most assemblers use the same conventions but I have seen .DW (define word, I guess) and .DS (define storage?). .WORD 0,0 is Ok as is .BYTE 0,1,2,3,4,5,6,7,8 etc. They both allow RAM vectors to be set, or byte tables, or word tables.

.ASC allows strings to be placed in memory such as:

.ASC "A STRING IN MEMORY"

Other than these few quicky psuedo ops, you will find we rarely use any of others, such as .FILE to chain in other source files, .BAS to write Basic code within your assembly listing, and great scads more. For a more detailed synopsis of PAL commands, look in Karl's 'Complete Inner Space Anthology'. You will find that most RAM-based assemblers can take advantage of our source listings.

TransBASIC, however, requires that modules be "merged" together. This is why they seem, at first glance, to have odd line number sequences. You'll notice that certain "areas" of each module are written in very specific line ranges. This is so they merge together with the same "areas" of other modules. If your assembler 1) uses the BASIC editor to create source code files, or 2) has a merge feature, you should have no problem after making the previously mentioned adjustments. Otherwise you may have to simulate the merging process.

Don't forget, The TransBASIC Disk is now available and comes complete with the SYMASS assembler. For \$9.95, TransBASIC becomes a totally self-contained utility. See our order card.

TransBASIC Installment #9

Nick Sullivan
Scarborough, Ont.

The TransBASIC Disk

The TransBASIC Disk contains all of the modules published so far and it comes with its own assembler, SYMASS 3.1. Any combination of modules can be linked into BASIC with only a few simple steps. From start to finish is usually no more than a couple of minutes. . . even less once you get the hang of it. It comes with a handy reference for just \$9.95. See the order card at center page.

TransBASIC Parts 1 to 8 Summary:

Part 1: *The concept of TransBASIC – a custom command utility that allows one to choose from a library only those commands that are necessary for a particular task.*

Part 2: *The structure of a TransBASIC module – each TransBASIC module follows a format designed to make them simple to create and “mergeable” with other modules.*

Part 3: *ROM routines used by TransBASIC – many modules make use of ROM routines buried inside the Commodore 64. Part 3 explains how to use these routines when creating new modules.*

Part 4: *Using Numeric Expressions – details on how to make use of the evaluate expression ROM routine.*

Part 5: *Assembler Compatibility – TransBASIC modules are written in PAL Assembler format. Techniques for porting them to another assembler were discussed here.*

Part 6: *The USE Command – The command ‘ADD’ merges TransBASIC modules into text space. However, as more modules are ADDED, merging gets slow. The USE command was written to speed things up. USE also counts the number of statements and functions USED and updates the totals (source line 95) automatically.*

Part 7 – *Usually TransBASIC modules don’t need to worry about interfering with one another. When two or more modules want to alter the same system vector, however, a potential crash situation exists. Part 7 deals with avoiding this problem.*

Part 8 – *Describes the five modules for Part 8.*

TransBASIC Part 9

This issue I want to do nothing more than present a few short modules that will bring this column into step with the new TransBASIC Disk.

First off (Program 1) is String Synthesis, which contains a handful of specialized functions for generating special strings. The most instructive of these from a module-creator’s point of view is the BUILD\$(function, which is a sort of glorified CHR\$(that can take multiple PETSCII arguments, including ranges, to build special strings.

The ability of BUILD\$(to handle multiple arguments means that we have to be careful in managing the memory used for intermediate results, since each argument can itself be a complex expression with its own function calls, theoretically even including calls to BUILD\$(itself. For this reason, BUILD\$(uses the two routines PSHTM and PULTEM, which together take care of saving and restoring the temporary memory registers T2 through T6 whenever a new argument is evaluated.

This might be a good time to mention a couple of things that distinguish statements from functions with respect to zero page storage. One is that the locations \$14 and \$15 are used by statements only, never by functions, so you can expect data store in those locations to survive expression evaluations intact, no matter how complex the expression may be. The POKE statement, for example, stores the POKE address at \$14/15 before evaluating the POKE value. Just remember never to use \$14/15 for storage when writing functions of your own, or you’ll end up clobbering some innocent statement that calls your function.

Another point is that both statements and functions have access to the TransBASIC storage area T2 through T6. If you use this area and then evaluate an expression, do not expect the registers to be unchanged. Either push the values onto the stack with the PSHTM routine, or one like it, or create a storage area within your own program code.

The Delay module (Program 2) contains a single statement, DELAY, which hangs the computer for a specified number of hundredths of seconds. One thing you might want to adopt for your own programs is the check for the STOP key (JSR \$A82C), which will break automatically to direct mode if the key is down.

The Slide module (Program 3) contains the statement SLIDE, which lets you move a sprite by specifying a displacement (relative) rather than a destination (absolute). SLIDE will wait until the raster scan is off the current location of the sprite before allowing it to move: in most cases this eliminates the shearing effect that arises when the raster catches a sprite in motion. Another interesting thing about SLIDE is that you can specify the direction of movement either as an integer or as a string. The routine that interprets the input might be useful in other commands as well.

The Make module (Program 4) contains the statement MAKE, which prints a specified number of repetitions of a string. You can use this to produce patterns and borders, and strings requiring repetitive cursor movement.

The Centre module (Program 5) contains the statement CENTRE, which prints a specified string of up to 40 characters centred on the monitor screen. The handy thing about this command is that it ignores control characters in the string (RVS and colour control characters, for example) when deciding how far to indent the string.

Finally, the Vocab Manager module (Program 6) contains two statements and two functions that will help in vocabulary searching applications like adventure games. The FILE statement, which is similar in structure to DATA, reads in alphanumeric strings and stores them under the BASIC ROM. With the SCAN(function, you can find the position of a particular string within the vocabulary.

When Vocab Manager is combined with other modules like Inline (in TransBASIC #8), First & BF\$ (#7), and Strip & Clean (#4), many applications that depend on input parsing become much simpler to program. Not only that, but the strings stored in the vocabulary are unknown to BASIC itself, and will not create garbage collection problems.

New Commands

DELAY (Type: Statement Cat #: 026)

Line Range: 3180-3214

Module: DELAY

Example: IF A = B THEN DELAY 100: PRINT "WHAT?"

Execution is suspended for the specified number of hundredths of seconds (0 to 65535). The timing is not accurate for very small values.

SLIDE (Type: Statement Cat #: 043)

Line Range: 3830-3928

Module: SLIDE

Example: FOR I = 1 TO 30: SLIDE 0, "E": NEXT

Example: SLIDE 3,2,84

Example: DI\$ = "U": SLIDE 7,DI\$,2

This command takes two arguments plus an optional third. The first is the sprite number (0-7), the second the direction in which it is to be displaced, and the third is the amount of

displacement. If the third argument is not present it is taken to be one. The second argument may be given as a number from 0 to 3; as a string beginning with one of "n", "e", "s", "w"; or as a string beginning with one of "u", "r", "d", "l". The strings may be in either upper or lower case.

MAKE (Type: Statement Cat #: 048)

Line Range: 4106-4142

Module: MAKE

Example: MAKE 22, "TRANSBASIC" + CHR\$(13)

The string argument is printed the specified number of times (up to 255) from the current cursor position.

CENTRE (Type: Statement Cat #: 049)

Line Range: 4144-4192

Module: CENTRE

Example: CENTRE "A PAINTED SHIP UPON A PAINTED OCEAN"

The string is centred on the current screen line. Control characters are ignored in calculating the offset from the margin. Strings longer than 40 characters (not counting control characters) generate a STRING TOO LONG error.

FILE (Type: Statement Cat #: 050)

Line Range: 4194-4272

Module: VOCAB MANAGER

Example: FILE "SWORD,MACE,SPEAR,POISON-TIPPED BANANA"

The strings separated by commas are stored under the BASIC ROM starting at \$A001 (40961). A pointer (FLPTR) points to the next free byte. Only alphanumerics are stored. Upper case alphabetic characters are converted to lower case. A vocabulary built by FILE statements can be searched with the SCAN(function (053). The only quote allowed in a FILE statement is the one that precedes the string data; also, no other statement may follow the FILE statement on the same line.

INITFP (Type: Statement Cat #: 051)

Line Range: 4274-4306

Module: VOCAB MANAGER

Example: INITFP

Example: INITFP 43257

The FILE statement pointer is initialized. If no parameter is present the pointer is initialized to address 40961. If a parameter between 40961 (\$A001) and 49151 (\$BFFF) is present the pointer is initialized to that address. The second form of the INITFP statement would normally be used only when a prepared vocabulary is loaded from disk, instead of being generated from FILE statements within a program. In this case the FILE statement pointer would have to be initialized to the value determined with the FPLOC function (052) after the vocabulary was first generated.

FPLOC (Type: Function Cat #: 052)

Line Range: 4308-4314

Module: VOCAB MANAGER

Example: PRINT FPLOC-40961

A quasi-variable returning the current value of the FILE statement pointer (40961 to 49151).

SCAN (Type: Function Cat #: 053)

Line Range: 4316-4468

Module: VOCAB MANAGER

Example: IF SCAN(AN\$)<83 GOTO 770

The vocabulary compiled by the FILE statement is searched for an entry matching the argument string. Only alphanumeric characters are converted to lower case. The number of the first matching vocabulary entry is returned, counting from one. Zero is returned if the search is unsuccessful.

ALPH\$ (Type: Function Cat #: 021)

Line Range: 2894-2900

Module: STRING SYNTHESIS

Example: PRINT LEFT\$(ALPH\$,13)

A quasi-variable that returns a string consisting of the lower case alphabet.

UCALPH\$ (Type: Function Cat #: 022)

Line Range: 2902-2908

Module: STRING SYNTHESIS

Example: PRINT ALPH\$ + UCALPH\$

A quasi-variable that returns a string consisting of the upper case alphabet.

NUM\$ (Type: Function Cat #: 023)

Line Range: 2910-2926

Module: STRING SYNTHESIS

Example: A = AWAIT(NUM\$)

A quasi-variable that returns a string consisting of the digits from 0 to 9.

RV\$ (Type: Function Cat #: 024)

Line Range: 2928-2984

Module: STRING SYNTHESIS

Example: PRINT RV\$("RUMPELSTILTSKIN")

Returns the argument string in reverse order (in this case, "NIKSTLITSLEPMUR").

BUILD\$ (Type: Function Cat #: 025)

Line Range: 2986-3098

Module: STRING SYNTHESIS

Example: A\$ = BUILD\$(36,48;57,32,65;70)

Returns a string specified by its ASCII components. Individual values may be specified, as well as ranges. In the latter case the low and high ends of the range are separated by a semicolon. The string "\$0123456789 ABCDEF" is returned by the example.

Program 1: STRING SYNTHESIS

OE	0 rem string synthesis (aug 29/84) :			
FH	1 :			
MH	2 rem 0 statements, 5 functions			
HH	3 :			
CF	4 rem keyword characters: 28			
JH	5 :			
NJ	6 rem keyword	routine	line	ser #
MI	7 rem f/alph\$	alph	2894	021
CB	8 rem f/ucalph\$	ucalph	2902	022

OE	9 rem f/num\$	num	2910	023
BK	10 rem f/rvs\$(	rvs	2928	024
MO	11 rem f/build\$(	build	2986	025
AI	12 :			
EN	13 rem u/pshtem (3100/060)			
KP	14 rem u/pultem (3134/061)			
HP	15 rem u/kpftop (3156/062)			
EI	16 :			
PD	17 rem =====			
GI	18 :			
LC	603 .asc " alph " :.byte\$a4:.asc " ucalph "			
	:.byte\$a4:.asc " num " :.byte\$a4			
CF	604 .asc " rvs\$ " :.byte\$a8:.asc " build\$ " :.byte\$a8			
KK	1603 .word alph-1,ucalph-1,num-1			
MB	1604 .word rvs-1,build-1			
OB	2894 ucalph lda # " A " ;range of upper			
EE	2896 ldx # " Z " ; case alphabet			
HH	2898 bne num1			
KM	2900 ;			
DG	2902 alph lda # " a " ;range of lower			
MM	2904 ldx # " z " ; case alphabet			
PH	2906 bne num1			
CN	2908 ;			
IB	2910 num lda # " 0 " ;range of digits			
OL	2912 ldx # " 9 "			
CM	2914 num1 sta t3			
PE	2916 lda #0			
EM	2918 sta t2			
DK	2920 lda #\$80			
MM	2922 sta t4			
AH	2924 bne bu2			
EO	2926 ;			
AJ	2928 rvs jsr \$aef4 ;eval expr, chk ')			
PK	2930 jsr \$b6a3 ;create descriptor			
CD	2932 rv1 sta \$61 ;save length			
KK	2934 stx t5 ;save pointer			
AH	2936 sty t6 ; to string			
EG	2938 jsr \$b47d ;allocate memory			
KE	2940 tay ;test string null			
EH	2942 beq rv3 ; yes			
CM	2944 dey ;index to last char			
IM	2946 lda #0 ;index to 1st chart			
MB	2948 sta t2 ;lower index save			
PL	2950 rv2 sty t3 ;upper index save			
FB	2952 lda (t5),y ;get upper char			
GK	2954 pha ;set it aside			
HB	2956 ldy t2 ;get lower index			
FC	2958 lda (t5),y ;get lower char			
LP	2960 tax ;set it aside			
GK	2962 pla ;re-get upper char			
PA	2964 sta (\$62),y ;store as lower			
NN	2966 txa ;re-get lower char			
DF	2968 ldy t3 ; and upper index			
NC	2970 sta (\$62),y ;store as upper			
OH	2972 beq rv3 ;when len(str) = 1			
HN	2974 inc t2 ;bump lower index			
AO	2976 dey ;back upper index			
HO	2978 cpy t2 ;test indices cross			
GD	2980 bcs rv2 ; not yet			
PG	2982 rv3 jmp \$b4ca ;return str descr			
OB	2984 ;			
HK	2986 build ldy #0 ;clear temp storage			
KG	2988 sty t2			
AH	2990 sty t4			
DL	2992 bu1 jsr pshtem ;push t2 - t6			
CG	2994 jsr kpf1 ;eval byte to .x			

OK	2996	stx \$67	; and save
GA	2998	jsr pultem	;pull t2 - t6
KK	3000	ldx \$67	;retrieve byte
HH	3002	stx t3	; and save
EL	3004	jsr \$79	;test range char
LA	3006	cmp #", "	
MJ	3008	bne bu2	; no
EA	3010	jsr pshtem	;push t2-t6
LJ	3012	jsr kpftop	;eval byte to .x
AM	3014	stx \$67	; and save
IB	3016	jsr pultem	;pull t2-t6
ML	3018	ldx \$67	;retrieve byte
HN	3020 bu2	txa	;test upper bound
BK	3022	sec	; >= lower bound
CA	3024	sbc t3	
EJ	3026	bcc bu7	; no
MO	3028	adc #0	;test rangesize 256
LM	3030	bcs bu8	; yes
DD	3032	pha	;push rangesize
LI	3034	adc t2	;test result > 255
BN	3036	bcs bu8	; yes
NE	3038	sta t2	;save result so far
PC	3040	pla	;pull rangesize
HE	3042	stx t3	;save upper bound + 1
OP	3044	jsr \$b47d	;reserve str space
LH	3046	stx \$22	;create pointer to
FH	3048	sty \$23	; string data
FE	3050	ldx t3	;get upper bound + 1
PD	3052	sta t3	;save string size
LO	3054	ldy #\$ff	;init index to str
DJ	3056 bu3	txa	;char to store
FJ	3058	iny	;bump index
KC	3060	cpy t3	;test = string size
NN	3062	beq bu4	; yes
MJ	3064	sta (\$62),y	;store character
JL	3066	dex	;next char down
NJ	3068 bcc	bu3	;branch always
DF	3070 bu4	bit t4	;test alph\$ etc
FO	3072	bmi bu6	; yes
OG	3074	jsr \$79	;test more to build
EC	3076	cmp #", "	
LO	3078	bne bu5	; no
FD	3080	jsr \$73	;skip comma
OM	3082	bne bu1	;branch always
EO	3084 bu5	jsr \$aef7	;check close paren
JG	3086 bu6	lda t2	;create descriptor
KP	3088	ldx \$62	
DA	3090	ldy \$63	
KF	3092	jmp rv1	;reverse the string
HP	3094 bu7	jmp \$b248	; 'illegal quantity'
OK	3096 bu8	jmp \$a571	; 'string too long'
AJ	3098 ;		
EA	3100 pshtem	lda #3	;check 6 stack
IC	3102	jsr \$a3fb	; bytes free
LH	3104	pla	;save return addr
NO	3106	sta \$71	
MK	3108	pla	
EP	3110	sta \$72	
NM	3112	ldx #4	;push t6 to t2
HC	3114 pht1	lda t2,x	
IK	3116	pha	
FO	3118	dex	
EH	3120	bpl pht1	
KG	3122 pht2	lda \$72	;retrieve rts addr
AL	3124	pha	
DM	3126	lda \$71	

EL	3128	pha	
GC	3130	rts	
CL	3132 ;		
PB	3134 pultem	pla	;save return addr
LA	3136	sta \$71	
KM	3138	pla	
CB	3140	sta \$72	
DD	3142	ldx #\$fb	;pull t2 to t6
FN	3144 plt1	pla	
GG	3146	sta \$7,x	
IC	3148	inx	
FI	3150	bmi plt1	
PK	3152	bpl pht2	;retrieve rts addr
IM	3154 ;		
AO	3156 kpftop	jsr \$73	;skip separator
HN	3158 kpf1	lda \$33	;push fretop ptr
EN	3160	pha	
IO	3162	lda \$34	
IN	3164	pha	
OA	3166	jsr \$b79e	;eval byte to .x
IM	3168	pla	;pull fretop ptr
OC	3170	sta \$34	
MO	3172	pla	
PC	3174	sta \$33	
EF	3176	rts	
AO	3178 ;		

Program 2: DELAY

HK	0 rem delay (aug 25/84)	:	
FH	1 :		
AI	2 rem 1 statement, 0 functions		
HH	3 :		
FO	4 rem keyword characters: 5		
JH	5 :		
NJ	6 rem keyword	routine	line ser #
JO	7 rem delay	dela	3180 026
MH	8 :		
HD	9 rem =====		
OH	10 :		
DH	106 .asc "delaY"		
IG	1106 .word dela-1		
KB	3180 dela	jsr \$ad8a	;eval num expr
CC	3182	jsr \$b7f7	;conv to int at \$14
OK	3184 de1	ldy #\$0e	;count 1/100 sec
MC	3186 de2	ldx #\$85	
NG	3188 de3	dex	
PF	3190	bne de3	
DD	3192	dex	
AG	3194	bne de2	
FN	3196	jsr \$a82c	;check stop key
IC	3198	ldx \$14	;decrement counter
MG	3200	bne de4	
PG	3202	ldy \$15	
FN	3204	beq de5	;countdown complete
OA	3206	dec \$15	
PG	3208 de4	dec \$14	
IG	3210	jmp de1	;not done yet
KK	3212 de5	rts	
EA	3214 ;		

Program 3: SLIDE

HL	0 rem slide (aug 25/84)	:
FH	1 :	


```

AI 2 rem 1 statement, 0 functions
HH 3 :
FO 4 rem keyword characters: 5
JH 5 :
NJ 6 rem keyword routine line ser #
JL 7 rem s/slide slid 3830 043
MH 8 :
DP 9 rem u/chkspr (3664/037)
CM 10 rem u/raschk (3676/038)
GL 11 rem u/direct (3930/044)
KN 12 rem d/powers (3694/039)
BI 13 :
MC 14 rem this module also contains one
NO 15 rem line from set sprites -- 3624
EI 16 :
PD 17 rem =====
GI 18 :
DI 110 .asc "slidE"
GJ 1110 .word slid-1
PB 3624 xs3 jmp $b248 ;'illegal quantity'
JO 3664 chkspr jsr $73 ;skip byte
OC 3666 chs1 jsr $b79e ;eval expr to .x
HL 3668 cpx #8 ;test valid sprite
OF 3670 bcs xs3 ; no
EE 3672 rts
AN 3674 ;
JD 3676 raschk pha
FO 3678 ras1 lda $d012 ;get raster pos'n
FG 3680 sbc $d001,x ;test above sprite
IF 3682 bcc ras2 ; yes
JB 3684 cmp #$2b ;test below sprite
EB 3686 bcc ras1 ; no
FN 3688 ras2 pla
GF 3690 rts
CO 3692 ;
FJ 3694 powers .byte 1,2,4,8,16,32,64,128
GO 3696 ;
HF 3830 slid jsr chs1 ;eval sprite #
MJ 3832 stx $14 ;save
MO 3834 jsr $aefd ;check for comma
LL 3836 jsr direct ;get direction
PA 3838 pha ;push direction
NN 3840 lda $14 ;push sprite #
OH 3842 pha
NI 3844 lda #1 ;save default
NN 3846 sta t3 ;displacement
EC 3848 jsr $79 ;test for comma
KC 3850 cmp #", "
EO 3852 bne sl1 ; no
KH 3854 jsr $b79b ;eval displacement
ON 3856 stx t3 ; and store
JF 3858 sl1 pla ;pull sprite #
BI 3860 tay ;mask index .y
NP 3862 asl ;position index .x
DO 3864 tax
PB 3866 pla ;pull direction
GE 3868 jsr raschk ;wait for raster
JD 3870 bne sl2 ;direction not up
AB 3872 lda $d001,x ;subtract disp
CC 3874 sbc t3 ; from y-pos'n
HF 3876 sta $d001,x
CB 3878 rts
HO 3880 sl2 cmp #2 ;test dir down
IA 3882 bne sl3 ; no

```

```

KE 3884 clc ;add disp
NA 3886 lda $d001,x ; to y-pos'n
EE 3888 adc t3
FG 3890 sta $d001,x
AC 3892 rts
EG 3894 sl3 cmp #1 ;test dir right
MB 3896 bne sl5 ; no
OJ 3898 lda $d000,x ;add disp
NN 3900 clc ; to y-pos'n
CF 3902 adc t3
CH 3904 sta $d000,x
IH 3906 bcc sl6 ;don't cross seam
LH 3908 sl4 lda $d010 ;toggle msb
GF 3910 eor powers,y ; of x-pos'n
OB 3912 sta $d010
GD 3914 rts
NG 3916 sl5 lda $d000,x ;subtract disp
NG 3918 sec ; from x-pos'n
CI 3920 sbc t3
EI 3922 sta $d000,x
JF 3924 bcc sl4 ;cross seam
LK 3926 sl6 rts
OM 3928 ;
LO 3930 direct jsr $ad9e ;eval direction
JA 3932 bit $0d ;test expr type
DD 3934 bmi di1 ; string
OB 3936 jsr $b7a1 ;eval numeric to .x
LK 3938 cpx #4 ;test < 4
KE 3940 bcs di5 ; no
IC 3942 txa ;return dir in .a
EF 3944 rts
MP 3946 di1 jsr $b6a6 ;create descriptor
KJ 3948 tay ;test length zero
MD 3950 beq di3 ; yes
CB 3952 ldy #0 ;get first char
ED 3954 lda ($22),y
CO 3956 ldy #$0f ;test valid dir
NG 3958 di2 cmp dirs,y
JE 3960 beq di4 ; yes
FD 3962 dey
MI 3964 bpl di2
EH 3966 di3 jmp $af08 ;'syntax'
KC 3968 di4 tya ;reduce to numeric
LF 3970 lsr
NF 3972 lsr
CH 3974 rts
BE 3976 di5 jmp $b248 ;'illegal quantity'
AA 3978 ;
CD 3980 dirs .asc "UuNnRrEeDdSsLIWw"
EA 3982 ;

```

Program 4: MAKE

```

AM 0 rem make (aug 25/84) :
FH 1 :
AI 2 rem 1 statement, 0 functions
HH 3 :
EO 4 rem keyword characters: 4
JH 5 :
NJ 6 rem keyword routine line ser #
IO 7 rem make mak 4106 048
MH 8 :
HD 9 rem =====
OH 10 :

```



```

LI 111 .asc "makE"
CF 1111 .word mak-1
FD 4106 mak jsr $b79e ;eval # repetitions
OC 4108 txa ;push
KI 4110 pha
CA 4112 jsr $aefd ;check for comma
MM 4114 jsr $ad9e ;eval string
BA 4116 jsr $b6a3 ;make descriptor
HJ 4118 tay ;string length
FC 4120 pla ;pull # repetitions
AH 4122 sta t3 ;countdown register
DN 4124 tya
PI 4126 mak1 ldx t3 ;get remaining reps
FD 4128 beq mak2 ;all done
OE 4130 dec t3 ;count down
AE 4132 pha ;print string
DG 4134 jsr $ab24
AL 4136 pla
JF 4138 jmp mak1
DN 4140 mak2 rts
EK 4142 ;

```

Program 5: CENTRE

```

FD 0 rem centre (sept 4/84) :
FH 1 :
AI 2 rem 1 statement, 0 functions
HH 3 :
GO 4 rem keyword characters: 6
JH 5 :
NJ 6 rem keyword routine line ser #
KG 7 rem centre cntr 4144 049
MH 8 :
HD 9 rem =====
OH 10 :
PO 112 .asc "centrE"
IL 1112 .word cntr-1
MP 4144 cntr jsr $ad9e ;eval string
PB 4146 jsr $b6a3 ;make descriptor
NF 4148 tay ;index from str end
HJ 4150 pha ;push string length
MG 4152 ldx #0 ;# printable chars
PK 4154 ce1 dey ;back up index
MA 4156 cpy #$ff ;test done
AA 4158 beq ce2 ; yes
CI 4160 lda ($22),y ;get a character
JJ 4162 and #$7f ;clear high bit
OM 4164 cmp #$20 ;test ctrl char
HM 4166 bcc ce1 ; yes
KD 4168 inx ;bump counter
PO 4170 bne ce1 ;branch always
FK 4172 ce2 txa ;test counter <= 40
PM 4174 sec
NK 4176 sbc #$29
MC 4178 bcs ce4 ; no
DJ 4180 eor #$ff ;negate and halve
CB 4182 lsr ; result
KM 4183 ldx $d3 ;test logical line
KJ 4184 cpx #$28 ; 40 or 80
IJ 4185 bcc ce3 ; 40
OP 4186 adc #$27 ;add 40 (carry set)
DP 4187 ce3 sta $d3 ;set cursor horiz
BL 4188 pla ;pull string length
PF 4189 jmp $ab24 ;print string

```

```

LP 4190 ce4 jmp $a571
GN 4192 ;

```

Program 6: VOCAB MANAGER

```

CL 0 rem vocab manager (aug 29/84) :
FH 1 :
JH 2 rem 2 statements, 2 functions
HH 3 :
CE 4 rem keyword characters: 20
JH 5 :
NJ 6 rem keyword routine line ser #
KM 7 rem s/file fil 4194 050
IE 8 rem s/initfp infptr 4274 051
JL 9 rem f/fploc fplo 4308 052
JD 10 rem f/scan( scan 4316 053
PH 11 :
OH 12 rem u/cifchr (2560/003)
PE 13 rem u/usfp (2620/006)
CN 14 rem u/cifnum (4092/047)
NN 15 rem d/flptr (4470/054)
EI 16 :
PD 17 rem =====
GI 18 :
DN 113 .asc "filEinitfP"
OB 610 .asc "fploCscan" :.byte$a8
PF 1113 .word fil-1,infptr-1
EC 1610 .word fplo-1,scan-1
HC 2560 cifchr cmp #$5b ;test alphabetic
OK 2562 bcc cic1 ; and if so return
LK 2564 clc ; carry set
FM 2566 bcc cic2
OB 2568 cic1 cmp #$41
FJ 2570 cic2 rts
CI 2572 ;
GK 2620 usfp ldx #0 ;convert .a/.y
AH 2622 stx $0d ; to floating point
DN 2624 sta $62 ; from unsigned
OO 2626 sty $63 ; 16-bit integer
ON 2628 ldx #$90
HM 2630 sec
NH 2632 jmp $bc49
AM 2634 ;
AE 4092 cifnum cmp # ":" ;test numeric
LM 4094 bcc cin1 ; and if so return
HK 4096 clc ; carry set
CO 4098 bcc cin2
KP 4100 cin1 cmp # "0"
NL 4102 cin2 rts
OH 4104 ;
NC 4194 fil cmp #$22 ;test leading quote
CD 4196 bne fi4 ; no
FM 4198 ldy flptr ;make pointer to
IJ 4200 lda flptr+1 ; first free byte
OI 4202 sty $22
DD 4204 sta $23
OM 4206 fi1 ldy #0 ;set up index
AP 4208 jsr $73 ;get a character
GP 4210 bcc fi2 ;numerics ok
HE 4212 cmp #0 ;test end of line
GE 4214 beq fi3 ; yes
MF 4216 cmp #$22 ;test embedded qte
NE 4218 beq fi4 ; yes
GH 4220 cmp # ",", ;test end of word

```


OE	4222	beq fi3	; yes	AA	4348	dec 1	;switch out basic
MN	4224	jsr cifchr	;test alphabetic	LB	4350	ldy #\$ff	
OB	4226	bcc fi1	; no	ND	4352	ldx #0	;init spare index
FD	4228 fi2	tax	;save byte	MC	4354 sca1	dec t4	;decr arg byte cntr
MB	4230	sta (\$22),y	;store to buffer	JI	4356 sca2	iny	
AK	4232	inc \$22	;bump pointer	GG	4358	cpy t3	;test end of word
HH	4234	bne fi1		DB	4360	beq sca4	; yes
JF	4236	inc \$23	;test end of buffer	BP	4362	lda (\$22),y	;get arg byte
GB	4237	lda \$23		IL	4364	jsr cifnum	;test numeric
GP	4238	cmp #\$c0	; (\$c000)	HB	4366	bcs sca3	; yes
FF	4240	bne fi1	; no	AI	4368	and #\$7f	;conv cap to lower
ON	4242	jmp \$a435	; 'out of memory'	OG	4370	jsr cifchr	;test alphabetic
PC	4244 fi3	pha	;push new byte	FN	4372	bcc sca1	; yes
HL	4246	dey	;point to previous	KO	4374 sca3	sta (\$62,x)	;add to new string
EP	4248	dec \$23	; byte in buffer	MP	4376	jsr bmp62	;bump new str ptr
AF	4250	txa	;get old byte	NN	4378	bne sca2	;branch always
MP	4252	ora #\$80	;set high bit	BD	4380 sca4	stx t5	;reset word counter
ED	4254	sta (\$22),y	;store to buffer	AO	4382	stx t6	
FC	4256	inc \$23	;fix pointer	FN	4384	bit t4	;test srch str null
MM	4258	pla	;pull new byte	KC	4386	bmi sca11	; yes
MI	4260	cmp #", "	;test comma	NC	4388	lda #1	;init vocab pointer
AH	4262	beq fi1	; yes	PN	4390	sta \$62	; to \$a001
LA	4264	ldy \$22	;store new pointer	OH	4392	lda #\$a0	
NF	4266	lda \$23	; value to flptr	JP	4394	sta \$63	
IL	4268	bne ifp4	;branch always	OJ	4396 sca5	inc t5	;bump word counter
LE	4270 fi4	jmp \$af08	; 'syntax error'	EC	4398	bne sca6	
GC	4272 ;			IH	4400	inc t6	
LO	4274 infptr	beq ifp2	;no param	BH	4402 sca6	lda \$63	;set carry if vocab
OI	4276	jsr \$ad8a	;eval param	FB	4404	cmp flptr + 1	; pointer at end
CL	4278	jsr \$b7f7	;convert to integer	MP	4406	bne sca7	; of buffer
OJ	4280	cmp #\$a0	;test >= \$a000	GM	4408	lda \$62	
JK	4282	bcc ifp1	; no	HF	4410	cmp flptr	
GN	4284	beq ifp3	; = is special case	HL	4412 sca7	txa	;.x = .y = 0
PK	4286	cmp #\$c0	;test < \$c000	NA	4414	tay	
IK	4288	bcc ifp4	; yes	EE	4416	bcs sca12	;end of buffer
OA	4290 ifp1	jmp \$b248	; 'illegal quantity'	PL	4418	dey	;set up pre-incr
DF	4292 ifp2	ldy #1	;default init	LF	4420 sca8	iny	;bump pointer
CD	4294	lda #\$a0	; to \$a001	DB	4422	lda (\$62),y	;get vocab byte
MF	4296 ifp3	cpy #0	;test = \$a000	FM	4424	cpy t4	;test last arg byte
HO	4298	beq ifp1	; yes	JG	4426	beq sca9	; yes
HM	4300 ifp4	sty flptr	;init flptr	DP	4428	cmp (\$24),y	;test arg = vocab
LM	4302	sta flptr + 1		JG	4430	beq sca8	; yes
ML	4304	rts		OP	4432	bne sca10	; no
IE	4306 ;			FA	4434 sca9	sbc (\$24),y	;test last vocab
GO	4308 fplo	ldy flptr	;get flptr value	DD	4436	cmp #\$80	; byte
FJ	4310	lda flptr + 1		GG	4438	beq sca11	; yes
OF	4312	jmp usfp	;return as fl. pt.	JP	4440 sca10	lda (\$62,x)	;advance vocab
AF	4314 ;			DI	4442	php	; pointer to
NH	4316 scan	jsr \$aef4	;eval str, test)	JF	4444	jsr bmp62	; end of word + 1
GL	4318	jsr \$b6a3	;get descriptor	CC	4446	plp	
HC	4320	sta t3	;store length	EJ	4448	bpl sca10	
EE	4322	sta t4		OO	4450	bmi sca5	;try next word
FD	4324	txa	;push data address	FE	4452 sca11	ldy t5	;get word counter
CG	4326	pha		OI	4454	lda t6	
PJ	4328	tya		AL	4456 sca12	inc 1	;switch in basic
GG	4330	pha		MJ	4458	jmp \$b391	;return as fl. pt.
MJ	4332	lda t3	;reserve memory	GA	4460 bmp62	inc \$62	;bump ptr at \$62/63
HC	4334	jsr \$b47d		IB	4462	bne b62	;return z clear
EE	4336	stx \$24	;make ptr to	BC	4464	inc \$63	
IN	4338	sty \$25	; reserved space	DF	4466 b62	rts	
NA	4340	pla	;make ptr to	KO	4468 ;		
LE	4342	sta \$23	; argument data	PA	4470 flptr	.word \$a001	;ptr to file bufr
AI	4344	pla		OO	4472 ;		
OL	4346	sta \$22					

Longer Life For Your 64 and 1541

Robert V. Davis
Salina, KS

With the price of Commodore 64 computers at one-fourth what it was when the machines first hit the market, the temptation to replace an older 64 with one of the later models is strong. But for those of us willing to break out the screwdriver and soldering iron, there are some minor improvements possible to prolong the life of our computers and disk drives. Note that all the following instructions will invalidate your warranty, if any, and anyone who is not comfortable with the idea of digging into electronic equipment should go to the next article.

The first modification to the Commodore 64 is to improve its video quality . . . this only applies to those of us who have the earlier model C-64s with a five pin video output jack. The addition of some of the luminance signal to the composite colour video will usually sharpen the picture noticeably on a colour monitor. Note Figure 1, the illustration of the top centre of the C-64 main printed circuit board.

Between the five pin video connector and the aluminum box containing the TV modulator will be a resistor (usually 470 ohms). As shown in Figure 1, this resistor will connect the solder pad labelled number two to the ground at the edge of the board, passing over 'point 1'. Using a small soldering iron (25 Watts), undo the ground end of the resistor. Then solder a 150-ohm quarter-watt resistor to point one. Attach the other end of that 150-ohm resistor to the still-connected end of the original resistor at point two. Then, probably using additional wire, re-attach the other end of the 470-ohm resistor to the ground, all the while avoiding solder bridges, bare wires touching each other, and so on.

By the way, if you are using a monochrome video monitor on your C-64, a better display results from taking the luminance output instead of the normal video output containing colour information to your video display. You will have to move a wire in the five-pin DIN connector from pin four to pin one. Again, both of these hints are appropriate to those older C-64s which have a five-pin video connector.

Some C-64s came from the factory without a heat sink attached to the five volt regulator chip which is mounted next to the joystick ports. That regulator supplies power to the video circuitry and runs rather hot. The addition of a heat sink, along with heat transfer compound will really help the regulator do its job. Look next to the joystick ports on the right side of the board for a small device on three legs soldered close to the corner of the video and system shield box. If a flat black slotted hunk of metal about one inch square is bolted or riveted to the regulator, you are OK. If not, the regulator will be sticking up with a hole through its top just begging you to help it cool down. The parts list suggests a possible heat sink which will probably require some bending before it will properly fit.

A couple of ways to keep your 1541 (or 2031LP) disk drive cooler are in order now. The hard work involves taking the drive completely apart and adding heat sink grease to help transfer heat from the big black heat sink at the back of the drive to the frame of the unit. I have found this helpful in both old and new 1541 drives, to help get the heat from the drive and twelve volt regulators spread around inside the unit, instead of concentrated near the 6502 and 6522 chips.

Note again, you will have to remove the plastic top of the drive, the RF shield on top of the main PC board, and then remove the PC board itself to get to the part of the heatsink where you can apply the heat transfer

compound. This is not for the faint of heart. Keep paper towels handy to clean up any heatsink grease other than that on the proper surfaces.

Finally, a foot or so of 5/16 inch wooden dowel rod cut into two to three inch lengths with four matching rubber feet can be (with some sanding) forced into the holes on the bottom of the 1541 where the screws holding the top of the case reside. (*Ends of pencils work great for this! -Ed.*) These legs will raise the drive and allow improved air circulation through it, prolonging the life of the electronic internals.

There you have some ways of keeping your system in good health at a minimum of expense. Good luck!

Parts list:
(Radio Shack Part numbers listed)

150 ohm 1/4 watt resistor	#271-1312
Heatsink	#276-1363
Heatsink grease	#276-1372
5/16 wooden dowel rod (4 two-inch lengths)	
5/16 inch rubber feet for ends of dowel	

(Four brand new pencils, cut to the proper length, may be used to replace the dowels and rubber feet)

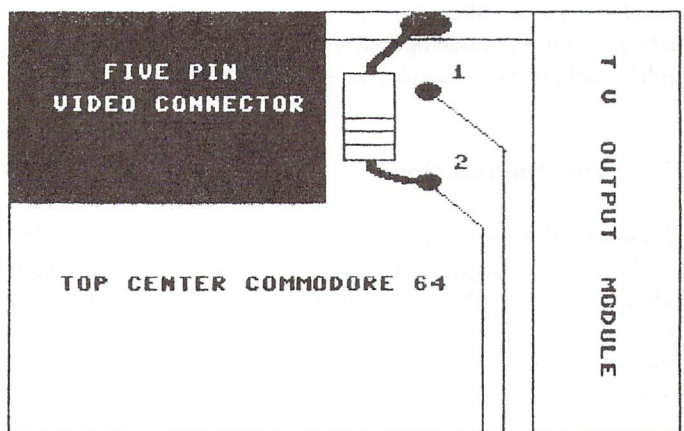


Figure 1

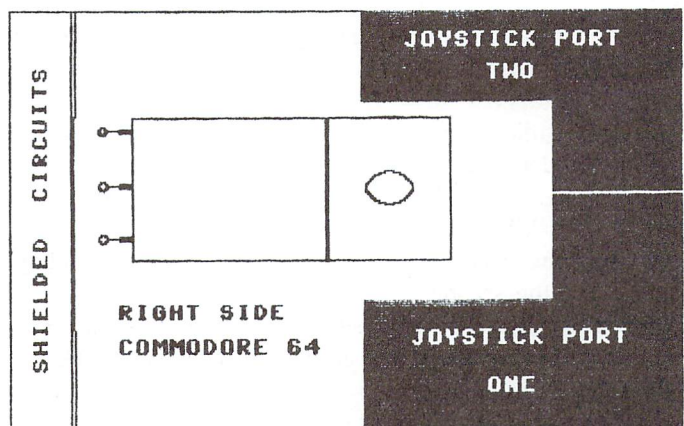


Figure 2

Machine language program to set one array equal to another.

Because operations in BASIC must be handled through the interpreter, some tasks can be very slow. For example, the following BASIC line can take a very long time:

```
for i = 1 to n:a(n) = b(n):next n
```

Many operating systems for larger computers have machine language (ML) routines that allow matrices or arrays (arrays are just one-dimension matrices) to be manipulated like non-dimensioned variables. In such a system, the above example could be written like `A = B`. For large, multi-dimensional arrays, or for repetitive operations, the time saved with such a utility could be very significant.

What The Program Does

To perform the operation in the above example, use SYS 51800 "A,B" with the array names in quotes (to avoid conflicts with other ML utilities that start at 49152, this program is moved up to just below where the BASIC wedge would be loaded). Any properly dimensioned arrays may be used. The only restrictions are that both arrays must be of the same type and dimensioned the same size. The utility will not check for this, only that both names are in the array table. Unlike BASIC, the subroutine will not automatically dimension an array. Instead, an error message is printed and the program halts if both arrays have not been dimensioned.

As stated earlier, any type of array can be used and the subroutine does recognize the difference between types. For example, `A,A%` and `A$` would be treated as different arrays just like in BASIC. When the subroutine returns to BASIC, each element of the first array will be the same as the corresponding element in the second array. The second array will not be affected. Note, the order in which the arrays are dimensioned is not important, only that the arrays are dimensioned before calling the routine. Also just to avoid possible confusion, I have been using A and B as my examples but there are no restric-

tions on the names that you can use, except for the normal BASIC restrictions.

How The Program Works

For those interested in ASSEMBLY programming, the commented listing in Program 1 should be useful. The listing is compatible with the IEA Assembler, but can be easily adapted to other Assemblers. In general, the following tasks are performed; ROM routines are used to find the location and length of the string in the calling SYS command. The two array names are then separated and the appropriate type designator is added. In BASIC, all arrays are stored in a block of memory. The starting address of this block is stored in locations \$30,\$2f. This address is loaded and the array memory area is scanned to find the starting address of the first array passed from BASIC. Some juggling of the array names then takes place so that the same portion of code can be used to find the address for the second name.

The length of the second array (actually the offset to the beginning of the next array in the storage area) is then found. The second array is then stored byte by byte in the first array. This simple byte for byte operation is why the routine is able to handle any type of variable. It is also the reason that the programmer must use similar arrays. If the first array was shorter than the second, the program would write past the end of the first array with possibly fatal results!

Typing It In

Program 2 is a ML loader program for the routine. As usual, first type it in and save it. With a disk in your drive, run the program. When the program is finished, you will have a file on your disk; "MATRIXML". To use the routine type `LOAD "MATRIXML",8,1` and `(RETURN)`. Then type `NEW,(RETURN)` and load your BASIC program.

To illustrate the speed of this routine type in and run the following program:

```
OK 100 dim b(1000),d(1000)
DF 110 print "beginning basic loop "
BG 120 t1 = ti
GI 130 for i = 0 to 1000:b(i) = d(i):next
IH 140 t2 = ti
CF 150 print "end of basic loop
AA 160 print "beginning ml loop
JJ 170 t3 = ti
BP 180 sys51800 "b,d"
AL 190 t4 = ti
DE 200 print "end of ml loop
PJ 210 print "basic = ";t2-t1;" jiffies
KL 220 print "ml = ";t4-t3;" jiffies
GA 230 stop
```

This program just sets D equal to C. The actual values in the array are of no interest in this case because we are only interested in the relative speeds of the two methods. On my machine, BASIC takes 425 jiffies and ML only 22. That means that the ML routine is 20 times faster than BASIC. If the arrays are changed to integer (C%,D%) the times are 429 and 9, making ML almost 50 times faster! Different size arrays would yield different time savings. For all but the very shortest arrays, the ML routine should provide a significantly faster operation.

Future Additions

At present, I am working to expand this routine into an entire matrix operations package. Some of the other operations that I am working on include:

1. Initialize the array – set the entire array equal to a user defined value.
2. Four function math operations – add, subtract, multiply or divide an array by a variable or by another array.
3. Find maximum or minimum values of an array.

Such a package would be useful in graphics or other programs where large arrays of data must be worked with. These additional operations will be included in a single program and be reached through different entry points.

Matrix Manipulator BASIC Loader

```
GA 30 hi = int(51800/256):lo = 51800-hi*256
BE 40 open 1,8,1,"0:matrixml"
HH 50 print#1,chr$(lo)chr$(hi);
KO 60 for i = 51800 to 52203
HF 70 read da:print#1,chr$(da);
EF 80 next
NC 90 close 1
FB 100 data 76, 117, 202, 128, 44, 144, 214, 65
PO 110 data 82, 82, 65, 89, 32, 78, 79, 84
EA 120 data 32, 68, 73, 77, 69, 78, 83, 73
AE 130 data 79, 78, 69, 68, 0, 32, 158, 173
AL 140 data 32, 163, 182, 134, 251, 132, 252, 170
DJ 150 data 160, 0, 140, 173, 2, 140, 171, 2
AL 160 data 177, 251, 141, 172, 2, 200, 202, 177
LJ 170 data 251, 201, 47, 144, 6, 141, 173, 2
LB 180 data 76, 141, 202, 201, 44, 240, 20, 201
BH 190 data 37, 208, 8, 173, 172, 2, 9, 128
LO 200 data 141, 172, 2, 173, 173, 2, 9, 128
GK 210 data 141, 173, 2, 200, 202, 177, 251, 201
NI 220 data 47, 144, 248, 141, 170, 2, 200, 202
FC 230 data 240, 32, 177, 251, 201, 47, 144, 6
JO 240 data 141, 171, 2, 76, 190, 202, 201, 37
DA 250 data 208, 8, 173, 170, 2, 9, 128, 141
PA 260 data 170, 2, 173, 171, 2, 9, 128, 141
OB 270 data 171, 2, 160, 3, 185, 139, 0, 153
OF 280 data 91, 202, 136, 16, 247, 169, 0, 160
IC 290 data 3, 153, 139, 0, 136, 208, 250, 141
LD 300 data 167, 2, 165, 48, 133, 252, 165, 47
KD 310 data 133, 251, 160, 2, 177, 251, 133, 253
MG 320 data 200, 177, 251, 133, 254, 160, 0, 177
DG 330 data 251, 205, 172, 2, 208, 8, 200, 177
BL 340 data 251, 205, 173, 2, 240, 62, 165, 252
IL 350 data 24, 101, 254, 133, 252, 165, 251, 24
LF 360 data 101, 253, 133, 251, 144, 2, 230, 252
IO 370 data 165, 50, 197, 252, 208, 6, 165, 49
FI 380 data 197, 251, 240, 3, 76, 2, 203, 162
DK 390 data 95, 160, 202, 134, 251, 132, 252, 160
JF 400 data 0, 177, 251, 240, 7, 32, 210, 255
OP 410 data 200, 76, 73, 203, 32, 226, 203, 162
AC 420 data 26, 76, 55, 164, 165, 251, 141, 168
FL 430 data 2, 165, 252, 141, 169, 2, 169, 1
NG 440 data 205, 167, 2, 240, 28, 173, 170, 2
FN 450 data 141, 172, 2, 173, 171, 2, 141, 173
EP 460 data 2, 173, 169, 2, 133, 140, 173, 168
MM 470 data 2, 133, 139, 238, 167, 2, 76, 250
OO 480 data 202, 173, 169, 2, 133, 142, 173, 168
GN 490 data 2, 133, 141, 160, 3, 177, 141, 133
PC 500 data 252, 136, 177, 141, 133, 251, 165, 142
DF 510 data 24, 101, 252, 133, 254, 165, 141, 24
IP 520 data 101, 251, 133, 253, 208, 2, 230, 254
IO 530 data 169, 4, 101, 141, 133, 141, 208, 2
JC 540 data 230, 142, 169, 4, 101, 139, 133, 139
AA 550 data 208, 2, 230, 140, 160, 0, 177, 141
PB 560 data 145, 139, 230, 141, 208, 2, 230, 142
CB 570 data 230, 139, 208, 2, 230, 140, 166, 141
DH 580 data 228, 253, 208, 234, 166, 142, 228, 254
GL 590 data 208, 228, 160, 3, 185, 91, 202, 136
IN 600 data 16, 250, 96, 0
```


Matrix Manipulator Source Code

```

100 ;ml routine to set a=b
110 ;where a,b are arrays
120 ;written by
130 ;richard richmond
140 ;308 rosewood ave
150 ;springfield, ohio 45506
160 ;(513)322-7650
170 * = $ca58 ;sys 51800 "a,b"
180 jmp start
190 zpage = *
200 * = ++4
210 two = $02aa
220 one = $02ac
230 chrout = $ffd2
240 word = *
250 .asc "array not dimensioned"
260 .byte 0
270 start = * ;use rom routine
280 jsr $ad9e ;to get string
290 jsr $b6a3
300 stx $fb ;address
310 sty $fc
320 tax ;length
330 ldy #$00
340 sty one+$01
350 sty two+$01
360 first = * ;first character
370 lda ($fb),y
380 sta one ;1st array
390 ab = *
400 iny
410 dex
420 lda ($fb),y
430 sta one+$01 ;2nd char
440 bcc skip
450 sta one+$01
460 jmp ab
470 skip = *
480 beq second ;comma
490 ;check for '$,%' in last character
500 ;name, bit 7 set in both bytes if
510 ;array is integer
520 ;bit 7 in 2nd byte set if
530 ;array is string
540 beq second
550 cmp #$25
560 bne skip1
570 lda one
580 ora #$80
590 sta one
600 skip1 = *
610 lda one+$01
620 ora #$80
630 sta one+$01
640 second = *
650 ;repeat for second argument
660 iny
670 dex
680 lda ($fb),y
690 cmp #$2f
700 bcc second
710 sta two
720 sec2 = *
730 iny
740 dex ;check for
750 beq done ;string end
760 lda ($fb),y
770 cmp #$2f ;check for
780 bcc ac ;alphanum
790 sta two+$01
800 jmp sec2
810 ac = *
820 cmp #$25
830 bne ad
840 lda two
850 ora #$80
860 sta two
870 ad = *
880 lda two+$01
890 ora #$80
900 sta two+$01
910 done = *
920 ;done with names, now save part
930 ;of zero page 8b-8e
940 ldy #$03
950 szpage = *
960 lda $008b,y
970 sta zpage,y
980 dey
990 bpl szpage
1000 lda #$00
1010 ldy #$03
1020 clear = *
1030 sta $008b,y

```

```

1040 dey
1050 bne clear
1060 sta $02a7 ;initial cntr
1070 done1 = *
1080 ;store address of beginning
1090 ;of array storage in fc,fb
1100 lda $30
1110 sta $fc
1120 lda $2f
1130 sta $fb
1140 d0 = *
1150 ldy #$02
1160 lda ($fb),y
1170 sta $fd ;offset lo
1180 iny
1190 lda ($fb),y
1200 sta $fe ;offset hi
1210 ldy #$00
1220 ;load 1st character of array
1230 ;and compare with 1st of
1240 ;argument
1250 lda ($fb),y
1260 cmp one
1270 bne d2
1280 iny
1290 lda ($fb),y
1300 cmp one+$01
1310 beq b3 ;jmp when
1320 ;array found
1330 d2 = *
1340 lda $fc
1350 clc
1360 ;add hi byte offset to address
1370 adc $fe
1380 sta $fc
1390 lda $fb
1400 clc
1410 ;add lo byte offset to address
1420 adc $fd
1430 sta $fb
1440 bcc d4
1450 inc $fc
1460 d4 = *
1470 ;check if end of array
1480 ;storage has been reached
1490 lda $32
1500 cmp $fc
1510 bne d3
1520 lda $31
1530 cmp $fb
1540 beq out
1550 ;branch to error routine
1560 ;if end of array storage
1570 d3 = *
1580 jmp d0
1590 ;jmp back and check next array
1600 out = * ;beginning of error routine
1610 ;print error message
1620 ldx #>word : ldy #,word
1630 stx $fb
1640 sty $fc
1650 ldy #$00
1660 error = *
1670 lda ($fb),y
1680 beq return
1690 jsr chrout ;rom routine
1700 iny
1710 jmp error
1720 return = *
1730 jsr reset ;restore zpage
1740 ldx #$1a
1750 jmp $a437 ;exit through
1760 ;rom routine to print error
1770 b3 = *
1780 ; store address of first array
1790 ; in argument
1800 lda $fb
1810 sta $02a8
1820 lda $fc
1830 sta $02a9
1840 b4 = *
1850 lda #$01 ;check
1860 cmp $02a7 ;pointer
1870 ; jmp to b5 2nd time
1880 ; through loop
1890 beq b5
1900 ; transfer second argument
1910 ; address to 1st storage
1920 lda two
1930 sta one
1940 lda two+$01
1950 sta one+$01
1960 ; transfer address of first
1970 ; argument to zero page
1980 ; at $8c,$8b
1990 lda $02a9

```

```

2000 sta $8c
2010 lda $02a8
2020 sta $8b
2030 inc $02a7
2040 ;increment pointer and
2050 ;branch back to find address
2060 ;of second argument
2070 jmp done1
2080 b5 = *
2090 ;address of second argument
2100 ;stored at $8e,$8d
2110 ;length of arrays stored
2120 ;at $fc,$fb
2130 ;only second array length
2140 ;used, both arrays must have
2150 ;dimensioned the same size
2160 lda $02a9
2170 sta $8e
2180 lda $02a8
2190 sta $8d
2200 ldy #$03
2210 lda ($8d),y
2220 sta $fc
2230 dey
2240 lda ($8d),y
2250 sta $fb
2260 lda $8e
2270 clc
2280 ;ending address of second
2290 ;array stored at
2300 ;$fe,$fd
2310 sta $fe
2320 lda $8d
2330 clc
2340 adc $fb
2350 sta $fd
2360 bne b7
2370 inc $fe
2380 b7 = *
2390 ;skip 4 bytes
2400 ;this skips past the name and
2410 ;offset bytes in array storage
2420 lda #$04
2430 adc $8d
2440 sta $8d
2450 bne b8
2460 inc $8e
2470 b8 = *
2480 lda #$04
2490 adc $8b
2500 sta $8b
2510 bne b9
2520 inc $8c
2530 b9 = *
2540 ;now ready to begin transferring
2550 ;data from 2nd array to 1st
2560 ldy #$00
2570 b13 = *
2580 ;load data byte by byte
2590 lda ($8d),y
2600 ;store in 1st array
2610 sta ($8b),y
2620 ;increment pointer - 2nd
2630 inc $8d
2640 bne b10
2650 inc $8e
2660 b10 = *
2670 ;increment pointer - 1st
2680 inc $8b
2690 bne b11
2700 inc $8c
2710 b11 = *
2720 ;compare pointer with end
2730 ;end of 2nd array
2740 ldx $8d
2750 cpx $fd
2760 bne b13
2770 ldx $8e
2780 cpx $fe
2790 bne b13
2800 reset = *
2810 ;when finished, restore
2820 ;zero page memory and return
2830 ;to basic
2840 ldy #$03
2850 restorez = *
2860 lda zpage,y
2870 dey
2880 bpl restorez
2890 rts
2900 .end

```


Jim Butterfield's Complete C128 Memory Map

A few issues back we published an abridged C128 RAM/ROM map as prepared by Jim Butterfield. At the time we were quite pleased to have the privilege of publication. Although the maps were not in any way complete, they were good enough to start many hungry programmers on their way with the C128.

After many months of careful and very well calculated pestering on our part, Jim has finally consented to allow us to publish his yet unreleased C128 Map. This opportunity comes as a form of prelude to Jim's yet unreleased new version of, "Machine Language For The Commodore 64 And Other Commodore Computers". Jim has carefully re-written it to include the C128, and as is usual with Jim's books, articles, videos, TV shows, etc., etc., etc., his Machine Language book takes the reader by the hand and gently force feeds knowledge without any painful infliction.

Jim's new book is expected to be released in April of 1986, published by Bradey, a division of Simon and Shuster. As with his last Machine Language book, this version will be available most everywhere through many of the major book stores. If after this incredible bit of JB propaganda you remain unmoved, let me assure you that I am not being paid for this, except for a bottle of Steam beer he bought me in San Francisco (for which I

paid him back promptly). If ever you get the chance, have a read. . . you will not be disappointed. - RTE

COMMODORE 128 Memory Maps

Jim Butterfield

These maps apply to the machine when used in the 128K mode. When used in the 64 mode, the machine's map is identical to that of the Commodore 64.

Architecture: "Bank numbers" as used in Basic BANK and the MLM addressing scheme are misleading; in fact, they are more correctly "configuration numbers". Bank 0 shows RAM level 0, which contains work areas and the user's Basic program. Bank 1 also shows RAM, this time (for addresses above hexadecimal 0400) level 1 which contains variables, arrays, and strings. Other "banks" are really configurations, with various types of ROM or I/O overlaying RAM. Thus, bank 15 (the most popular) is ROM and I/O covering RAM bank 0. Bank 14, however, is ROM and the character generator overlaying RAM bank 0. Architecture is set so that addresses below \$0400 reference bank 0 only. Other bank switching (more complex than the simplified 16-bank concept) is accomplished via storing a mask to address \$FF00, or calling up pre-stored masks by writing to \$FF01-\$FF04.

The Commodore C128 Memory Map as of February 1986

All Banks:								
Hex	Decimal	Description						
0000	0	I/O directional register	0076	118	Graphics flag	00D7	215	40/80 columns: 0 = 40 columns
0001	1	I/O port, similar to C64	0077	119	Color source number	00D8	216	Graphics mode code
0002-0004	2-4	SYS address, MLM registers (SR, PC)	0078-0079	120-121	Temporary counters	00D9	217	Character base: 0 = ROM, 4 = RAM
0005-0009	5-9	SYS, MLM register save (A, X, Y, SR/SP)	007A-007C	122-124	DS\$ descriptor	00DA-00DF	218-223	Misc work area
000A	10	Scan-quotes flag	007D-007E	125-126	BASIC pseudo-stack pointer	00E0-00E1	224-225	Pointer to screen line/cursor
000B	11	TAB column save	007F	127	Flag: 0 = direct mode	00E2-00E3	226-227	Color line pointer
000C	12	0 = LOAD, 1 = VERIFY	0080-0081	128-129	DOS, USING work flags	00E4	228	Current screen bottom margin
000D	13	Input buffer pointer/number of subscripts	0082	130	Stack pointer save for errors	00E5	229	Current screen top margin
000E	14	Default DIM flag	0083	131	Graphic color source	00E6	230	Current screen left margin
000F	15	Type: FF = string; 00 = numeric	0084	132	Multicolor 1 (1)	00E7	231	Current screen right margin
0010	16	Type: 80 = integer; 00 = floating point	0085	133	Multicolor 2 (2)	00E8-00E9	232-233	Input cursor log (row, column)
0011	17	DATA scan/LIST quote/memory flag	0086	134	Graphic foreground color (13)	00EA	234	End-of-line for input pointer
0012	18	Subscript/FNx flag	0087-008A	135-138	Graphic scale factors, X & Y	00EB	235	Position of cursor on screen line
0013	19	0 = INPUT; \$40 = GET; \$98 = READ	008B-008F	139-143	Graphic work values	00EC	236	Row where cursor lives
0014	20	ATN sign/Comparison evaluation flag	0090	144	Status word ST	00ED-00EE	237-238	Maximum screen lines, columns
0015	21	Current I/O prompt flag	0091	145	Keyswitch IA: STOP and RVS flags	00EF	239	Current I/O character
0016-0017	22-23	Integer value	0092	146	Timing constant for tape	00F0	240	Previous character printed
0018	24	Pointer: temporary string stack	0093	147	Work value, monitor, LOAD/SAVE	00F1	241	Character color
0019-0023	25-35	Stack for temporary strings	0094	148	Serial output: deferred character flag	00F2	242	Temporary color save
0024-0027	36-39	Utility pointer area	0095	149	Serial deferred character	00F3	243	Screen reverse flag
0028-002C	40-44	Product area for multiplication	0096	150	Cassette work value	00F4	244	0 = direct cursor; else programmed
002D-002E	45-46	Pointer: start-of-BASIC (for bank 0)	0097	151	Register save	00F5	245	Number of INSERTS outstanding
002F-0030	47-48	Pointer: start-of-variables (bank 1)	0098	152	How many open files	00F6	246	255 = Auto Insert enabled
0031-0032	49-50	Pointer: start-of-arrays	0099	153	Input device, normally 0	00F7	247	Text mode lockout
0033-0034	51-52	Pointer: end-of-arrays	009A	154	Output CMD device, normally 3	00F8	248	0 = Scrolling enabled
0035-0036	53-54	Pointer: string-storage (moving down)	009B-009C	155-156	Tape parity, output-received flag	00F9	249	Bell disable
0037-0038	55-56	Utility string pointer	009D	157	I/O messages: 192 = all, 64 = errors, 0 = nil	00FA-00FF	250-255	Not used
0039-003A	57-58	Pointer: limit-of-memory (bank 1)	009E-009F	158-159	Tape error pointers	0100-01FF	256-511	Processor stack area
003B-003C	59-60	Current BASIC line number	00A0-00A2	160-162	Jiffy Clock HML	0100-013E	256-318	Tape error log
003D-003E	61-62	Textpointer: BASIC work point	00A3-00AB	163-171	I/O work bytes	0100-0124	256-292	DOS work area
003F-0040	63-64	Utility Pointer	00AC-00AD	172-173	Pointer: tape buffer, scrolling	0125-0138	293-312	PRINT/USING work area
0041-0042	65-66	Current DATA line number	00AE-00AF	174-175	Tape end adds/End of program	0200-02A0	512-672	BASIC input buffer
0043-0044	67-68	Current DATA address	00B0-00B1	176-177	Tape timing constants	02A2-02AE	674-686	Bank peek subroutine
0045-0046	69-70	Input vector	00B2-00B3	178-179	Pointer: start of tape buffer	02AF-02BD	687-701	Bank poke subroutine
0047-0048	71-72	Current variable name	00B4-00B6	180-182	RS-232, Misc work values	02BE-02CC	702-716	Bank compare subroutine
0049-004A	73-74	Current variable address	00B7	183	Number of characters in file name	02CD-02E2	717-738	JSR to another bank
004B-004C	75-76	Variable pointer for FOR/NEXT	00B8	184	Current logical file	02E3-02FB	739-763	JMP to another bank
004D-004E	77-78	Y-save; op-save; BASIC pointer save	00B9	185	Current secondary address	02FC-02FD	764-765	Function execute hook [4C78]
004F	79	Comparison symbol accumulator	00BA	186	Current device	0300-0301	768-769	Error message link
0050-0055	80-85	Miscellaneous work area, pointers, and so on	00BB-00BC	187-188	Pointer to file name	0302-0303	770-771	BASIC warm start link
0056-0058	86-88	Jump vector for functions	00BD-00C5	189-197	I/O work pointers	0304-0305	772-773	Crunch BASIC tokens link
0059-0062	89-98	Miscellaneous numeric work area	00C6-00C7	198-199	Banks: I/O data, filename	0306-0307	774-775	Print tokens link
0063	99	Accum#1: exponent	00C8-00CB	200-203	RS-232 input/output buffer addresses	0308-0309	776-777	Start new BASIC code link
0064-0067	100-103	Accum#1: mantissa	00CC-00CD	204-205	Keyboard decode pointer (bank 15)	030A-030B	778-779	Get arithmetic element link
0068	104	Accum#1: sign	00CE-00CF	206-207	Print string work pointer	030C-030D	780-781	Crunch FE hook
0069	105	Series evaluation constant pointer	00D0	208	Number of characters in keyboard buffer	030E-030F	782-783	List FE hook
006A-006F	106-111	Accum#2: exponent, and so on	00D1	209	Number of programmed chars waiting	0310-0311	784-785	Execute FE hook
0070	112	Sign comparison, Acc#1 versus #2	00D2	210	Programmed key character index	0312-0313	786-787	Unused
0071	113	Accum#1 lo-order (rounding)	00D3	211	Key shift flag: 0 = no shift	0314-0315	788-789	IRQ vector [FA65]
0072-0073	114-115	Cassette buffer len/Series pointer	00D4	212	Key code: 88 if no key	0316-0317	790-791	Break interrupt vector [B003]
0074-0075	116-117	Auto line number increment	00D5	213	Key code: 88 if no key	0318-0319	792-793	NMI interrupt vector [FA40]
			00D6	214	Input from screen/from keyboard	031A-031B	794-795	OPEN vector [EFBD]

031C -031D	796-797	CLOSE vector [F188]	0A0F -0A17	2575-2583	RS-232 work values	1214 -1217	4628-4631	DO work pointers
031E -031F	798-799	Set-input vector [F106]	0A18	2584	RS-232 receive pointer	1218 -121A	4632-4634	USR program jump [7D28]
0320 -0321	800-801	Set-output vector [F14C]	0A19	2585	RS-232 input pointer	121B -121F	4635-4639	RND seed value
0322 -0323	802-803	Restore I/O vector [F226]	0A1A	2586	RS-232 transmit pointer	1222	4642	Sound tempo
0324 -0325	804-805	Input vector [EF06]	0A1B	2587	RS-232 send pointer	122F	4655	Music sequencer
0326 -0327	806-807	Output vector [EF79]	0A1D -0A1F	2588-2590	Sleep countdown; FFFF = disable	1234 -1237	4660-4663	Note image
0328 -0329	808-809	Test-STOP vector [F66E]	0A20	2592	Keyboard buffer size	1239 -123E	4665-4670	Current env pattern
032A -032B	810-811	GET vector [EEEE]	0A21	2593	Screen freeze flag	123F -1270	4671-4720	Envelope tables ..
032C -032D	812-813	Abort I/O vector [F222]	0A22	2594	Key repeat: 128 = all, 64 = none	123F -1248	4671-4680	AD(SR) pattern
032E -032F	814-815	Machine Lang Monitor link	0A23	2595	Key repeat timing	1249 -1252	4681-4690	(AD)SR pattern
0330 -0331	816-817	LOAD link	0A24	2594	Key repeat pause	1253 -125C	4691-4700	Waveform pattern
0332 -0333	818-819	SAVE link	0A25	2595	Graphics/text toggle latch	125D -1266	4701-4710	Pulse width pattern
0334 -0335	820-821	Control code (low) link	0A26	2596	40-col cursor mode	1267 -1270	4711-4720	Pulse width hi pattern
0336 -0337	822-832	High ASCII code link	0A27 -0A2A	2597-2600	40-col blink values	1271 -1274	4721-4274	Note: xx,xx,volume
0338 -0339	824-825	ESC sequence link	0A2B	2601	80-col cursor mode	1275	4725	Previous volume image
034A -0353	842-851	Keyboard buffer	0A2C	2602	40-col video \$D018 image	1276 -1278	4726-4728	Collision IRQ task table
0354 -035D	852-861	Tab stop bits	0A2E -0A2F	2606-2607	80 col pages - screen, color	1279 -127E	4729-4734	Collision IRQ address tables
035E -0361	862-865	Line wrap bits	0A40 -0A5A	2624-2650	40/80 pointer swap \$E0-FA	127F	4735	Collision mask
0362 -036B	866-875	Logical file table	0A60 -0A6D	2656-2669	40/80 data swap \$354-361	1280	4736	Collision work value
036C -0375	876-885	Device number table	0AC0	2752	PAT counter	12B1	4785	PEN work value
0376 -037F	886-895	Secondary address table	0AC1 -0AC4	2753-2756	ROM Physical Address Table	1300 -17FF	4864-6143	Unused
0380 -039E	896-926	CHRGOT subroutine	0B00 -0BBF	2816-3007	Cassette buffer	1800 -1BFF	6144-7167	Reserved for key functions
039F -03AA	927-938	Fetch from RAM bank 0	0BC0 -0BFF	3008-3071		1C00 -FBFF	7168-64511	BASIC RAM memory (text)
03AB -03B6	939-950	Fetch from RAM bank 1	0C00 -0DFF	3072-3583	RS-232 input, output buffers	1C00 -1FFF	7168-8186	Video (color) matrix (hi-res)
03B7 -03BF	951-959	Fetch from RAM bank 1	0E00 -0FFF	3584-4095	System sprites (56-63)	1FF8 -1FFF	8187-8191	Sprite identities (hi-res)
03C0 -03C8	960-968	Fetch from RAM bank 0	1000 -1009	4096-4105	Programmed key lengths	2000 -3FFF	8192-16383	Screen memory (hi-res)
03C9 -03D1	969-977	Fetch from RAM bank 0	100A -10FF	4106-4351	Programmed key definitions	4000 -FBFF	16384-64511	BASIC RAM memory (hi-res)
03D2 -03D4	978-980	Unused	1100 -1130	4352-4400	DOS Command staging area	Bank 1:	0400 -FBFF	1024-64511 Basic variables, arrays, strings
03D5	981	Current BANK for SYS, PEEK	1131 -116E	4401-4462	Graphics work area	Bank 14: Same as Bank 15, below, except:	D000 -DFFF	53248-57343 Character generator ROM
03D6 -03D9	982-985	INSTR work values	116F	4463	Trace mode: FF = on	Bank 15:	4000 -CFFF	16384-53247 ROM: BASIC
03DA	986	Bank location for string	1170 -1173	4464-4467	Renumbering pointers	D000 -D02E	53248-53294	40-col video chip 8564
03DB -03DD	987-989	Sprite work bytes	1174 -1177	4468-4471	Directory work pointers	D400 -D41C	54272-54300	SID sound chip 6581
03DF	991	Accum*1: Overflow	1178 -1197	4472-4473	Graphics index			Memory Management Unit 8722
03E0 -03E1	992-993	Sprite work bytes	117A -117B	4474-4475	Float-fixed vector [849F]	D500	54528	MMU primary config register
03E2	994	Graphic/Text backgrounds	117C -117D	4476-4477	Fixed-float vector [793C]	D501 -D504	54529-54532	MMU preconfig registers
03E3	995	Graphic/Multi color log	117E -11D5	4478-4565	Sprite motion tables (8 x 11 bytes)	D505 -D506	54533-54534	MMU mode, ram registers
03F0 -03F6	1008-1014	DMA link code	11D6 -11E5	4566-4581	Sprite X/Y positions	D507 -D50A	54535-54538	MMU page 0, page 1 regs
FF00	65280	MMU configuration register	11E6	4582	Sprite X-high positions	D600 -D601	54784-54785	80-column CRT contr 8563
FF01		Bank 0	11E7 -11E8	4583-4584	Sprite bump masks (sprite, backgnd)	10 -11	16-17	X, Y positions
FF02		Bank 1	11E9 -11EA	4585-4586	Light pen values, X and Y	12 -13	18-19	On-chip RAM address
FF03		Bank 14	11EB	4587	CHRGEN ROM page, text [D8]	1A	26	Background color
FF04		Bank 14 over RAM 1	11EC	4588	CHRGEN ROM page, graphics [D0]	1F	31	On-chip RAM data
FF01 -FF04	65281-65284	MMU load config registers	11ED	4589	Secondary address for RECORD	D800 -D8E7	55296-56295	Color nybbles
Bank 0:			11EE -11FF	4590-4607	Unused	DC00 -DC0F	56320-56336	CIA 1 (IRQ) 6526
0400 -07E7	1024-2023	40-column screen memory	1204 -1207	4612-4615	PU characters (, , \$)	DD00 -DD0F	56576-56591	CIA 2 (NMI) 6526
07F8 -07FF	2040-2047	Sprite identity area (text)	120B -120C	4619-4620	TRAP address: FFFF if none	DF00 -DF0A	57088-57098	DMA slot
0800 -09FF	2048-2560	BASIC pseudo-stack	1210 -1211	4624-4625	End of Basic (Bank 0)	E000 -FEFF	57344-65279	ROM: Kernal
0A0C	2572	CIA 1 interrupt log	1212 -1213	4626-4627	Basic program limit [FF00]	FF05 -FFFF	65285-65535	ROM: Transfer, Jump Table
0A0D	2573	CIA 1 timer enabled						

ROM Map

4000	Basic Entry Jumps	4B3F	Execute/Trace Statement	528F	Perform [data/bend]	5A1D	Put Sub To B-Stack	610A	Perform [key]
4009	Basic Restart	4BCB	Perform [stop]	529D	Perform [rem]	5A3D	Perform [go]	61A8	Perform [paint]
4023	Basic Cold Start	4BCD	Perform [end]	52A2	Scan To Next Stmt	5A60	Perform [cont]	627C	Check Painting Split
4045	Set-Up Basic Constants	4BF7	Setup FN Reference	52A5	Scan To Next Line	5A9B	Perform [run]	62B7	Perform [box]
4112	Chime	4C86	Evaluate <or>	52C5	Perform [if]	5ACA	Perform [restore]	642B	Perform [sshape]
417A	Set Preconfig Registers	4C89	Evaluate <and>	5320	Search/Skip Begin/Bend	5AF0	Keywords To Renumber	658D	Perform [gshape]
4189	Registers For \$D501	4CB6	Evaluate <compare>	537C	Skip String Constant	5AF8	Perform [renumber]	668E	Perform [circle]
418D	Init Sprite Movement Tabs	4D2A	Print 'ready'	5391	Perform [else]	5BAE	Renumber - Continued	6750	Draw Circle
419B	Print Startup Message	4D37	Error or Ready	53A3	Perform [on]	5BFB	Renumber Scan	6797	Perform [draw]
4251	Set Basic Links	4D3A	Print 'out of memory'	53C6	Perform [let]	5D19	Convert Line Number	67D7	Perform [char]
4267	Basic Links	4D3C	Error	54F6	Check String Location	5D68	Get Renumber Start	6955	Perform [locate]
4279	Chrgot For \$0380	4DAF	Break Entry	553A	Perform [print*]	5D75	Count Off Lines	6960	Perform [scale]
42CE	Get From (\$50) Bank 1	4DC3	Ready For Basic	5540	Perform [cmd]	5D89	Add Renumber Inc	69E2	Perform [color]
42D3	Get From (\$3F) Bank 1	4DE2	Handle New Line	555A	Perform [print]	5D99	Scan Ahead	6A4C	Color Codes
42D8	Get From (\$52) Bank 1	4F4F	Rechain Lines	5600	Print Format Char	5DA7	Set Up Block Move	6A5C	Log Current Colors
42DD	Get From (\$5C) Bank 0	4F82	Reset End-of-Basic	5612	Perform [get]	5DC6	Block Move Down	6A79	Perform [scnclr]
42E2	Get From (\$5C) Bank 1	4F93	Receive Input Line	5635	Getkey	5DDF	Block Move Up	6B06	Fill Memory Page
42E7	Get From (\$66) Bank 1	4FAA	Search B-Stack For Match	5648	Perform [input*]	5DEE	Check Block Limit	6B17	Set Screen Color
42EC	Get From (\$61) Bank 0	4FFE	Move B-Stack Down	5662	Perform [input]	5DF9	Perform [for]	6B30	Clear Hi-Res Screen
42F1	Get From (\$70) Bank 0	5017	Check Memory Space	569C	Prompt & Input	5E87	Perform [delete]	6B5A	Perform [graphic]
42F6	Get From (\$70) Bank 1	5047	Copy B-Stack Pointer	56A9	Perform [read]	5EFB	Get Line Number Range	6BC9	Perform [bank]
42FB	Get From (\$50) Bank 1	5050	Set B-Stack Pointer	57F4	Perform [next]	5F34	Perform [pudef]	6BD7	Perform [sleep]
4300	Get From (\$61) Bank 1	5059	Move B-Stack Up	587B	Perform [dim]	5F4D	Perform [trap]	6C09	Multiply Sleep Time
4305	Get From (\$24) Bank 0	5064	Find Basic Line	5885	Perform [sys]	5F62	Perform [resume]	6C2D	Perform [wait]
430A	Crunch Tokens	50A0	Get Fixed Pt Number	58B4	Perform [tron]	5FB7	Reinstate Trap Point	6C4F	Perform [sprite]
43E2	Check Keyword Match	50E2	Perform [list]	58B7	Perform [troff]	5FD8	Syntax Exit	6CB3	Bit Masks
4417	Keywords	5123	List Subroutine	58BD	Perform [rreg]	5FDB	Print 'can't resume'	6CC6	Perform [movspr]
46FC	Action Vectors	51D6	Perform [new]	5901	Assign <mid\$>	5FE0	Perform [do]	6DE1	Perform [play]
47D8	Function Vectors	51F3	Set Up Run	5975	Perform [auto]	6039	Perform [exit]	6E02	Analyze Play Character
4828	Defunct Vectors	51F8	Perform [clr]	5986	Perform [help]	608A	Perform [loop]	6EB2	Set SID Sound
4846	Unimplemented Commands	5238	Clear Stack & Work Area	59AC	Insert Help Marker	60B4	Print 'loop not found'	6EFD	Play Error
484B	Messages	5250	Pudge Characters	59CF	Perform [gosub]	60B7	Print 'loop without do'	6F03	Dotted Note
4A82	Find Message	5254	Back Up Text Pointer	59DB	Perform [goto]	60DB	Eval While/Until Argument	6F07	Note Length Char
4B34	Update Continue Pointer	5262	Perform [return]	5A15	Undef'd Statement	60E1	Define Programmed Key	6F1E	Note A-G

6F52	.. voxum ..	864D	Pull String Parameters	928D	Call 'plot'	B3C7	Print 'error'	C854	Chr\$(29) Cursor Right
6F69	Sharp	8668	Evaluate <len>	9293	Call 'get'	B3DB	Perform [f]	C85A	Chr\$(17) Cursor Down
6F6C	Flat	866E	Exit String Mode	9299	Make Room For String	B406	Perform [a.]	C875	Chr\$(157) Cursor left
6F78	Rest	8677	Evaluate <asc>	92EA	Garbage Collection	B536	Print 'space <esc-q>'	C880	Chr\$(14) Text
6FD7	Perform [tempo]	8688	Calc String Vector	9409	Evaluate <cos>	B57C	Check 2 A-Matches	C8A6	Chr\$(11) Lock
6FE4	Voice Times Two	869A	Set Up String	9410	Evaluate <sin>	B57F	Check A-Match	C8AC	Chr\$(12) Unlock
6FE7	Length Characters	874E	Build String to Memory	9459	Evaluate <tan>	B58B	Try Next Op Code	C8B3	Chr\$(19) Home
6FEC	Command Characters	877B	Evaluate String	9485	Trig Series	B599	Perform [d]	C8BF	Chr\$(146) Clear Rvs Mode
702F	Chime Seq	87E0	Clean Descriptor Stack	94B3	Evaluate <atan>	B5B1	Print '<cr> <esc-q>'	C8C2	Chr\$(18) Reverse
7039	SID Voice Steps	87F1	Input Byte Parameter	94E3	Series	B5D4	Display Instruction	C8C7	Chr\$(2) Underline-On
7046	Perform [filter]	8803	Params For Poke/Wait	9520	Print Using	B5F5	Print '<3 spaces>'	C8CE	Chr\$(130) Underline-Off
70C1	Perform [envelope]	8815	Float/Fixed	99C1	Evaluate <instr>	B659	Classify Op Code	C8D5	Chr\$(15) Flash-On
7164	Perform [collision]	882E	Subtract From Memory	9B0C	Evaluate <rdot>	B6A1	Get Mnemonic Char	C8DC	Chr\$(143) Flash-Off
7190	Perform [sprcolor]	8831	Evaluate <subtract>	9B30	Draw Line	B6C3	Mode Tables	C8E3	Open Screen Space
71B6	Perform [width]	8845	Add Memory	9BFB	Plot Pixel	B715	Mode Characters	C91B	Chr\$(20) Delete
71C5	Perform [vol]	8848	Evaluate <add>	9C49	Examine Pixel	B721	Compacted Mnemonics	C932	Restore Cursor
71EC	Perform [sound]	8917	Trim FAC*1 Left	9C70	Set Hi-Res Color Cell	B7A5	Input Parameter	C94F	Chr\$(9) Tab
72CC	Perform [window]	894E	Round Up FAC*1	9CCA	Video Matrix Lines Hi	B7CE	Read Value	C961	Chr\$(24) Tab Toggle
7335	Perform [boot]	895D	Print 'overflow'	9CE3	Position Pixel	B88A	Number Bases	C96C	Find Tab Column
7372	Perform [sprdef]	899C	Log Series	9D1C	Bit Masks	B88E	Base Bits	C980	Esc-z Clear All Tabs
7691	Sprite Vectors	89CA	Evaluate <log>	9D24	Calc Hi-Res Row/Column	B892	Display 5-Digit Address	C983	Esc-y Set Default Tabs
76EC	Perform [sprsav]	8A0E	Add 0.5	9DF2	Restore Pixel Cursor	B8A5	Display 2-Digit Byte	C98E	Chr\$(7) Bell
77B3	Perform [fast]	8A24	Multiply By Memory	9E2F	Parse Graphics Command	B8A8	Print Space	C9B1	Chr\$(10) Linefeed
77C4	Perform [slow]	8A27	Evaluate <multiply>	9E32	Get Color Source Param	B8AD	Print Cursor-Up	C9BE	Analyze Esc Sequence
77D7	Type Match Check	8A89	Unpack ROM to FAC*2	9F29	Conv Words Hi	B8B4	New Line	C9DE	Vectors
77DA	Confirm Numeric	8AB4	Unpack RAM1 to FAC*2	9F3D	Conv Words Lo	B8B9	Blank New Line	CA14	Esc-t Top
77DD	Confirm String	8AE3	Adjust FAC*1/*2	A022	Move Basic to \$1C01	B8C2	Output 2-Digit Byte	CA16	Esc-b Bottom
77E7	Print 'type mismatch'	8B17	Multiply By 10	A07E	Perform [catalog/directory]	B8D2	Byte to 2 Ascii	CA1B	Set Window Part
77EA	Print 'formula too complex'	8B2E	+ 10	A11D	Perform [dopen]	B8E7	Get Input Char	CA24	Exit Window
77EF	Evaluate Expression	8B33	Print 'division by zero'	A134	Perform [append]	B8E9	Get Character	CA3D	Esc-i Insert Line
78D7	Evaluate Item	8B38	Divide By 10	A157	Find Spare SA	B901	Copy Add0 to Add2	CA52	Esc-d Delete Line
793C	Fixed-Float	8B49	Divide Into Memory	A16F	Perform [dclose]	B90E	Calculate Add2-Add0	CA76	Esc-q Erase End
7950	Eval Within Parens	8B4C	Evaluate <divide>	A18C	Perform [dsave]	B922	Subtract	CA8B	Esc-p Erase Begin
795C	Check Comma	8BD4	Unpack ROM to FAC*1	A1A4	Perform [dverify]	B93C	Subtract 1	CA9F	Esc-@ Clr Remainder of Scrn
796C	Syntax Error	8BF9	Pack FAC*1 to \$5E	A1A7	Perform [dload]	B950	Increment Pointer	CABC	Esc-v Scroll Up
7978	Search For Variable	8BFC	Pack FAC*1 to \$59	A1C8	Perform [bsave]	B960	Decrement Pointer	CACA	Esc-w Scroll Down
7A85	Unpack RAM1 to FAC*1	8C00	Pack FAC*1 to RAM1	A218	Perform [bload]	B974	Copy to Register Area	CAE2	Esc-l Scroll On
7AAF	Locate Variable	8C28	FAC*2 to FAC*1	A257	Perform [header]	B983	Calculate Step/Range	CAE5	Esc-m Scroll Off
7B3C	Check Alphabetic	8C38	FAC*1 to FAC*2	A2A1	Perform [scratch]	B9B1	Perform [s+&%]	CAEA	Esc-c Cancel Auto Insert
7B46	Create Variable	8C47	Round FAC*1	A2D7	Perform [record]	BA07	Convert to Decimal	CAED	Esc-a Auto Insert
7CAB	Set Up Array	8C57	Get Sign	A322	Perform [dclear]	BA47	Transfer Address	CAF2	Esc-s Block Cursor
7D25	Print 'bad subscript'	8C65	Evaluate <sgn>	A32F	Perform [collect]	BA5D	Output Address	CAFE	Esc-u Underline Cursor
7D28	Print 'illegal quantity'	8C68	Byte Fixed-Float	A346	Perform [copy]	BA90	Perform [a]	CB0B	Esc-e Cursor Non Flash
7E3E	Compute Array Size	8C75	Fixed-Float	A362	Perform [concat]	C000	-cint-	CB21	Esc-f Cursor Flash
7E71	Array Pointer Subtrn	8C84	Evaluate <abs>	A36E	Perform [rename]	C006	Get From Keyboard	CB37	Esc-g Bell Enable
8000	Evaluate <fre>	8C87	Compare FAC*1 to Memory	A37C	Perform [backup]	C009	Screen Input Link	CB3A	Esc-h Bell Disable
8020	Decrypt Message	8CC7	Float-Fixed	A3BF	Parse DOS Commands	C00C	Screen Print Link	CB3F	Esc-r Screen Reverse
804A	Evaluate <val>	8CFB	Evaluate <int>	A5E7	Print 'missing file name'	C00F	-screen-	CB48	Esc-n Screen Normal
8052	String to Float	8D22	String to FAC*1	A5EA	Print 'illegal device number'	C012	-scnkey-	CB52	Esc-k End-of-Line
8076	Evaluate <dec>	8DB0	Get Ascii Digit	A5ED	Print 'string too long'	C018	-plot-	CB58	Get Screen Char/Color
80C5	Evaluate <peek>	8E17	Conversion Values	A627	DOS Command Masks	C021	Define FN Key	CB74	Check Screen Line of Lo
80E5	Perform [poke]	8E26	Print 'in'...	A7E1	Print 'are you sure?'	C024	IRQ Link	CB81	Extend/Trim Screen Line
80F6	Evaluate <err\$>	8E32	Print Integer	A80D	Release String	C027	Upload 80 Col	CB9F	Set Up Line Masks
8139	Swap .x With .y	8E42	Float to Ascii	A845	Set Bank 15	C02A	Swap 40/80	CBB1	Esc-j Start-of-Line
8142	Evaluate <hex\$>	8F76	+0.5	A84D	IRQ Work	C02D	Set Window	CBBC	Find End-of-Line
816B	Byte to Hex	8F7B	Decimal Constants	AA1F	Perform [stash]	C033	Screen Address Low	CBED	Move Cursor Right
8182	Evaluate <ngr>	8F9F	TI Constants	AA24	Perform [fetch]	C04C	Screen Address High	CC00	Move Cursor Left
818C	Get Graphics Mode	8FB7	Evaluate <sqr>	AA29	Perform [swap]	C065	I/O Link Vectors	CC1E	Save Cursor
819B	Evaluate <rchr>	8FBE	Raise to Memory Power	AE64	Encrypted Message	C06F	Keyboard Shift Vectors	CC27	Print Space
8203	Evaluate <joy>	8FC1	Evaluate <power>	AF00	Basic Vectors	C07B	Initialize Screen	CC2F	Print Character
824D	Evaluate <pot>	8FFA	Evaluate <negate>	B000	Perform [monitor]	C142	Reset Window	CC32	Print Fill Color
82AE	Evaluate <pen>	9005	Exp Series	B009	Break Entry	C150	Home Cursor	CC34	Put Char to Screen
82FA	Evaluate <pointer>	9033	Evaluate <exp>	B00C	Print 'break'	C156	Goto Left Border	CC5B	Get Rows/Columns
831E	Evaluate <rsprite>	90D0	I/O Error Message	B021	Print 'call' entry	C15C	Set Up New Line	CC6A	Read/Set Cursor
8361	Evaluate <rsprcolor>	90D8	Basic 'open'	B03D	Print 'monitor'	C17C	Do Screen Color	CCA2	Define Function Key
837C	Evaluate <bump>	90DF	Basic 'chout'	B050	Perform [r]	C194	(IRQ) Split Screen	CD2C	Esc-x Switch 40/80
8397	Evaluate <rspos>	90E5	Basic 'input'	B053	Print 'pc str. ...'	C234	Get a Key	CD57	Position 80-col Cursor
83E1	Evaluate <xor>	90EB	Redirect Output	B08B	Get Command	C29B	Input From Screen	CD6F	Set Screen Color
8407	Evaluate <rwindow>	90FD	Redirect Input	B0BC	Error	C2BC	Read Screen Char	CD9F	Turn Cursor On
8434	Evaluate <rnd>	9112	Perform [save]	B0BF	Print '?'	C2FF	Check For Quotes	CDCA	Set CRTC Register 31
8490	Rnd Multiplier	9129	Perform [verify]	B0E3	Perform [x]	C30C	Wrap Up Screen Print	CDCC	Set CRTC Register
849A	Value 32768	912C	Perform [load]	B0E6	Commands	C320	Ascii to Screen Code	CDD8	Read CRTC Register 31
849F	Float-Fixed Unsigned	918D	Perform [open]	B0FC	Vectors	C33E	Check Cursor Range	CDDA	Read CRTC Register
84A7	Evaluate Fixed Number	919A	Perform [close]	B11A	Read Banked Memory	C363	Do New Line	CDE6	Set CRTC to Screen Address
84AD	Float-Fixed Signed	91AE	Get Load/Save Parameters	B12A	Write Banked Memory	C37C	Insert a Line	CDF9	Set CRTC to Color Address
84C9	Float (.y.a)	91DD	Get Next Byte Value	B13D	Compare Banked Memory	C3A6	Scroll Screen	CE0C	Set Up 80 Column Char Set
84D0	Evaluate <pos>	91E3	Get Character or Abort	B152	Perform [m]	C3DC	Delete a Line	CE4C	Ascii Color Codes
84D9	Check Direct	91EB	Move to Next Parameter	B194	Perform [.]	C40D	Move Screen Line	CE5C	System Color Codes
84DD	Print 'illegal direct'	91F6	Get Open/Close Params	B1AB	Perform [x]	C4A5	Clear a Line	CE6C	Bit Masks
84E0	Print 'undef'd function'	9243	Release I/O String	B1CC	Print 'esc-o, up'	C53C	Set 80-column Counter to 1	CE74	40-Col Init Values (\$E0)
84E5	Set Up 16 Bit Fix-Float	9251	Call 'status'	B1D6	Perform [x]	C53E	Set 80-column Counter	CE8E	80-Col Init Values (\$0A40)
84F5	Print 'direct mode only'	9257	Call 'setlfs'	B1DF	Perform [i]	C55D	Keyboard Scan Subtrn	CEA8	Prog Key Lengths
84FA	Perform [def]	925D	Call 'setnam'	B1E8	Display Memory	C651	Key Pickup & Repeat	CEB2	Prog Key Definitions
8528	Check FN Syntax	9263	Call 'getin'	B20E	Print '<crvs-on>'	C6DD	Programmed Keys	E000	Reset Code
853B	Perform [fin]	9269	Call 'chout'	B231	Perform [c]	C6E7	Flash 40 Column Cursor	E04B	MMU Set Up Bytes
85AE	Evaluate <str\$>	926F	Call 'clrchn'	B234	Perform [t]	C72D	Print to Screen	E056	-restor-
85BF	Evaluate <chr\$>	9275	Call 'close'	B2C3	Add 1 to Op 3	C77D	Esc-o (escape)	E05B	-vector-
85D6	Evaluate <left\$>	927B	Call 'clail'	B2C6	Do Next Address	C79A	Vectors	E073	Vectors to \$0314
860A	Evaluate <right\$>	9281	Print Following Text	B2CE	Perform [h]	C7B6	Print Control Char	E093	-ramtas-
861C	Evaluate <mid\$>	9287	Set Load/Save Bank	B337	Perform [lsv]	C802	Print Hi-Bit Char	E0CD	Code For High RAM Banks

E105	RAM Bank Masks	E68E	Set RS-232 Bit Count	EEA8	IRQ Vectors	F53E	-save-	F7AE	Get Char From Memory
E109	-joinit-	E69D	(NMI) RS-232 Receive	EEB0	Kill Tape Motor	F5B5	Terminate Serial Input	F7BC	Store Loaded Byte
E1DC	Set Up CRTC Registers	E75F	Send to RS-232	EEB7	Check End Address	F5BC	Print 'saving'	F7C9	Read Byte to be Saved
E1F0	Check Special Reset	E795	Connect RS-232 Input	EEC1	Bump Address	F5C8	Save to Tape	F7D0	Get Char From Memory Bank
E242	Reset to 64/128	E7CE	Get From RS-232	EEC8	(IRQ) Clear Break	F5F8	-udtim-	F7DA	Store Char to Memory Bank
E24B	Switch to 64 Mode	E7EC	Interlock RS-232/Serial	EED0	Control Tape Motor	F63D	Watch For RUN or Shift	F7E3	Compare Char With Memory Bank
E263	Code to \$02	E805	(NMI) RS-232 Transmit I/O	EEEB	-getin-	F65E	-rdtim-	F7EC	Load Mem Control Mask
E26B	Scan All ROMs	E850	RS-232 Timings	EF06	-chrin-	F665	-settim-	F7F0	Bank Masks
E2BC	ROM Addresses Hi	E878	(NMI) RS-232 Receive Timing	EF48	Get Char From Tape	F66E	-stop-	F800	Subtrns to \$02A2-\$02FB
E2C0	ROM Banks	E8A9	(NMI) RS-232 Transmit Timing	EF79	-chrout-	F67C	Print 'too many files'	F85A	DMA Code to \$03F0
E2C4	Print 'cbrn' Mask	E8D0	Find Any Tape Header	EFBD	-open-	F67F	Print 'file open'	F867	Check Auto Start ROM
E2C7	VIC 8564 Set Up	E919	Write Tape Header	F0B0	Set CIA to RS-232	F682	Print 'file not open'	F890	Check For Boot Disk
E2F8	CRTC 8563 Set Up Pairs	E980	Get Buffer Address	F0CB	Check Serial Open	F685	Print 'file not found'	F90B	Print 'booting'
E33B	-talk-	E987	Get Tape Buffer Start & End Addr	F106	-chkin-	F688	Print 'device not present'	F92F	Print '...'
E33E	-listen-	E99A	Find Specific Header	F14C	-chkout-	F68B	Print 'not input file'	F98B	Wind Up Disk Boot
E43E	-acptr-	E9BE	Bump Tape Pointer	F188	-close-	F68E	Print 'not output file'	F9B3	Read Next Boot Block
E4D2	-second-	E9C8	Print 'press play ...'	F1E4	Delete File	F691	Print 'missing file name'	F9FB	To 2-Digit Decimal
E4E0	-tksa-	E9DF	Check Tape status	F202	Search For File	F694	Print 'illegal device no'	FA08	Block Read
E503	-ciout- Print Serial	E9E9	Print 'press record..'	F212	Set File Parameters	F697	Error #0	FA15	Print '#i'
E515	-untilk-	E9F2	Initiate Tape Read	F222	-clall-	F6B0	Messages	FA17	Print a Message
E526	-unlsn-	EA15	Initiate Tape Write	F226	-clrchn-	F71E	Print If Direct	FA40	NMI Sequence
E535	Reset ATN	EA26	Common Tape Code	F23D	Clear I/O Path	F722	Print I/O Message	FA65	(IRQ) Normal Entry
E545	Set Clock High	EA7D	Wait For Tape	F265	-load-	F731	-setnam-	FA80	Keyboard Matrix Un-Shifted
E54E	Set Clock Low	EA8F	Check Tape Stop	F27B	Serial Load	F738	-setlfs-	FAD9	Keyboard Matrix Shifted
E557	Set Data High	EEA1	Set Read Timing	F32A	Tape Load	F73F	Set Load/Save Bank	FB32	Keyboard Matrix C-Key
E560	Set Data Low	EAEB	(IRQ) Read Tape Bits	F3A1	Disk Load	F744	-rdst-	FB8B	Keyboard Matrix Control
E569	Read Serial Lines	EC1F	Store Tape Chars	F3EA	Burst Load	F757	Set Status Bit	FBE4	Keyboard Matrix Caps Lock
E573	Stabilize Timing	ED51	Reset Pointer	F48C	Close Off Serial	F75C	-setmsg-	FF00	MMU Controls
E59F	Restore Timing	ED5A	New Char Set Up	F4BA	Get Serial Byte	F75F	Set Serial Timeout	FF05	NMI Transfer Entry
E5BC	Prepare For Response	ED69	Send Transin to Tape	F4C5	Receive Serial Byte	F763	-memtop-	FF17	IRQ Transfer Entry
E5C3	Fast Disk Off	ED8B	Write Data to Tape	F503	Toggle Clock Line	F772	-membot-	FF33	Return From Interrupt
E5D6	Fast Disk On	ED90	(IRQ) Tape Write	F50C	Print 'u0' Disk Reset	F781	-iobase-	FF3D	Reset Transfer Entry
E5FB	Fast Disk On/Off	EE2E	(IRQ) Tape Leader	F50F	Print 'searching'	F786	Search For SA	FF47	Jumbo Jump Table
E5FF	(NMI) Transmit RS-232	EE57	Wind Up Tape I/O	F521	Send File Name	F79D	Search & Set Up File	FFFA	Transfer Vectors
E64A	RS-232 Handshake	EE9B	Switch IRQ Vector	F533	Print 'loading'	F7A5	Trigger DMA		

8502 Processor I/O Registers

0000	X	0 = in	1 = out	0 = in	1 = out	1 = out	1 = out	1 = out	00000
0001	X	Caps Key	Tape Motor	Tape Sense	Tape Output	HiRes	LoRes	Color Access	00001

8722 Memory Management Unit

D500	RAM select 0-3	HIGH RAM /ROM	MID RAM /ROM	LO RAM	C GEN	54528
D501-D504	Preconfiguration registers; Similar to D500, above					54529-54532
D505	40/80 Key	C64 Mode	Cartt-Sense Color-Bank	Fast Disk	X X Z80	54533
D506	Video-Bank	X	X	Shared RAM hi low	Shared RAM 0=1K	54534
D507	Zero Page Pointer (\$0000)					54535
D508						54536
D509	Stack Page Pointer (\$0000)					54537
D50A						54538

6526 CIA 1 (IRQ)

(Same as CIA 1 for C64, until DC0C)

DC00	Paddle Select A	Fire	Right	Joystick 0 Left	Down	Up	PRA 56320
DC01	Keyboard Row Select (inverted)						PRB 56321
DC02	Keyboard Column Read						DDRA 56322
DC03	\$FF - All Output						DDRB 56323
DC04	\$00 - All Input						TAL 56324
DC05	Timer A						TAH 56325
DC06	Timer B						TBL 56326
DC07							TBH 56327
DC0C	Serial (shift) Register						56332
DC0D	IRQ	X	X	Flag	S.Reg	X Tim.B	56333
DC0E	S Reg I/O		Load	O/S	Timer A Toggle	Start	56334
DC0F			Load	O/S	Timer B	Start	56335

DMA Controller

DF00	Busy	Fault	X	X	X	X	X	X	57088	
DF01	Exec	Sum	X	X	IRQ	Inc	Mode		57089	
DF02	Host Address								L	57090
DF03									H	57091
DF04	Expansion Address								L	57092
DF05									H	57093
DF06	X	X	X	X	X	Expansion Bank			57094	
DF07	Transfer Length								L	57095
DF08									H	57096
DF09	Checksum								57097	
DF0A	Version, Maximum-Memory								57098	

6526 CIA 2 (NMI)

(Same as CIA 2 for C64)

DD00	Serial IN	Clock IN	Serial OUT	Clock OUT	ATN OUT	RS232 OUT	Video	Block	PRA 56576
DD01	DSR IN	CTS IN		DCD* IN	RI* IN	DTR OUT	RTS OUT	RS232 IN	PRB** 56577
DD02	IN	IN	OUT	OUT	OUT	OUT	OUT	OUT	DDRA 56578
DD03	\$06 for RS232								DDRB 56579
DD04	Timer A								TAL 56580
DD05									TAH 56581
DD06	Timer B								TBL 56582
DD07									TBH 56583
DD0D	RS232 IN		Timer B		Timer A				ICR 56589
DD0E					Timer A Start				CRA 56590
DD0F					Timer B Start				CRB 56591

* Connected but not used by O.S.

** PRB is the Parallel User Port
DDRA = \$3F at reset

8564 Video Chip Control & Miscellaneous Registers

D011	Extended Clr. Mode		Bit Map	Display Enable	Row Select	Y-Scroll		53265	
D012	Raster Register							53266	
D013	Light Pen Input							X	53267
D014								Y	53268
D016	x	x	Reset	Multi Colour	Column Select	X-Scroll		53270	
D018	Screen VM13 VM12 VM11 VM10				Character Base CB13 CB12 CB11			x	53272
D019	IRQ	Interrupt Sense:			Light Pen	Spr-Spr Collision	Spr-Back Collision	Raster	53273
D01A	Interrupt Enable:			Light Pen	Spr-Spr Collisions	Spr-Back Collisions	Raster	53274	
Colour Registers									
D020	X				Exterior Colour (Border)				53280
D021	X				Background Colour #0				53281
D022	X				Background Colour #1				53282
D023	X				Background Colour #2				53283
D024	X				Background Colour #3				53284
D025	X				Sprite MultiColour #0				53285
D026	X				Sprite MultiColour #1				53286
D02F	x	x	x	x	x	[Keyboard Rows]		53295	
D030	X	X	X	X	X	X	Test	Fast Clock	53296

6581 SID Sound Chip (Identical to 6581 on C64)

Voice 1	Voice 2	Voice 3		Voice 1	Voice 2	Voice 3
D400	D407	D40E	Frequency	L	54272	54279 54286
D401	D408	D40F		L	54273	54280 54287
D402	D409	D410	Pulse Width	L	54274	54281 54288
D403	D40A	D411	0 0 0 0	H	54275	54282 54289
D404	D40B	D412	NSE Voice Type: PUL SAW TRI Key		54276	54283 54290
D405	D40C	D413	Attack Time: 2ms-8sec Decay Time: 6ms-24sec		54277	54284 54291
D406	D40D	D414	Sustain Level: Release Time: 6ms-24sec		54278	54285 54292
Voices are "write-only"						
D415	0 0 0 0 0	L	54293			
D416	Filter Frequency	H	54292			
D417	Resonance	Ext	54295			
D418	Passband BP LO	Master Volume	54296			
Filter and Volume (write only)						
D419	Paddle X (A/D #1)	54297				
D41A	Paddle Y (A/D #2)	54298				
D41B	Noise 3 (random)	54299				
D41C	Envelope 3	54300				
Sense (read only)						

Note: Special Voice Features
(TEST, RING, MOD, SYNC)
are omitted from the above diagram

8564 Video Chip Sprite Registers

Sprite 0 ↓ D000	Sprite 7 ↓ D00E	Sprite Registers			Sprite 0 ↓ 53248	Sprite 7 ↓ 53262
D001	D00F	X Position			53249	53263
		Y Position				
D027	D02E	Sprite Colour			53287	53294
Bit For Sprite#:						
7 ↓	6 ↓	5 ↓	4 ↓	3 ↓	2 ↓	1 ↓
D010	X-Position High					53264
D015	Sprite Enable Flags					53269
D017	Y-Expand					53271
D01B	Background Priority					53275
D01C	Sprite MultiColour Mode					53276
D01D	X-Expand					53277
D01E	Interrupt: Sprite Collision					53278
D01F	Interrupt: Background Collision					53279

8563 80-Column CRT Controller

D600 read (status):										
D600	Status	Light Pen	Vert Blank	X	X	X	X	X	54784	
D600 54784	D601 54785								Typical Value	
0 \$00	Horizontal Total								126	
1 \$01	Horizontal Characters Displayed (80)								80	
2 \$02	Horizontal Sync position								102	
3 \$03	Vertical Sync Width				Horizontal Sync Width				1 and 3	
4 \$04	X	Vertical Total						32 or 39		
5 \$05	X	X	X	Vertical Total Adjust					0	
6 \$06	X	Vertical Displayed (25)						25		
7 \$07	X	Vertical Sync Position						29 or 32		
8 \$08	X	X	X	X	X	X	Interlace	0		
9 \$09	X	X	X	Scan Lines per Character				7		
10 \$0A	X	Cursor Mode		Cursor Start					32	
11 \$0B	X	X	X	Cursor End					7	
12 \$0C	X	X	Display Address					H	0	
13 \$0D								L	0	
14 \$0E			Cursor Address					H	0	
15 \$0F								L	0	
16 \$10			Light Pen Input					H	varies	
17 \$11								L	varies	
18 \$12			Video RAM Address (See register 31)					H	varies	
19 \$13								L	varies	
20 \$14			Colour Address					H	8	
21 \$15								L	0	
22 \$16	Character Total				Character Display Horizontal				120	
23 \$17	X	X	X	Character Display Vertical					8	
24 \$18	Block Copy	Scrn RVS	Blink Rate	V Scroll					32	
25 \$19	Bit Map	Colour Enable	Semi Graph	Wide Pixel	H Scroll				64 or 71	
26 \$1A	Foreground Colour				Background Colour				240	
27 \$1B	Scroll Control Horizontal								0	
28 \$1C	Char Set Address			RAM	X	X	X	X	32	
29 \$1D	X	X	X	Underline Scan Line Count					7	
30 \$1E	Character Count								varies	
31 \$1F	Video RAM data (see registers 18,19)								varies	
32 \$20	Block Copy Start Address							H	varies	
33 \$21								L	varies	
34 \$22	Display Enable							begin	125	
35 \$23								end	100	
36 \$24	X	X	X	X	DRAM Refresh Rate				5	

The C128 – You Can Bank On It

Jim Butterfield
Toronto, Ontario

You may have noticed that the Commodore 128 has sixteen “memory banks”. In Basic, you may call whatever bank you want (for PEEK, POKE, SYS or WAIT) by using the BANK command with a value from 0 to 15. Similarly, machine language types will reference banks in the monitor by prefixing an address with a digit from 0 to F – the same bank values of 0 to 16.

However, the average programmer – with no cartridge, internal ROM, or RAM expansion – can only make use of four of these numbers. The only ones that make sense are banks 0, 1, 14 and 15 (hex 0, 1, E, and F)

What about the other numbers? Banks 2 and 3 are reserved for memory expansion. Banks 4 to 7 and bank 12 are only useful if the empty socket inside your machine has been fitted with an “internal” ROM chip. Banks 5 to 11 and 13 are only useful if a cartridge ROM is plugged into your machine. And even if you have these extra things fitted, chances are that a commercial software house has taken care of all the banking you’re likely to need, leaving you with little to look at for fun.

I don’t like the term ‘bank’ as it is used on this machine. These numbers represent configurations; each so-called bank is an assembly of varying parts of memory.

Only ‘Bank 0’ is not a mixture: it uses one kind of memory only, the RAM where your Basic programs are held (usually called RAM 0). All the others are mixtures of different types of memory appearing at various addresses. Even bank 0 is slightly “impure” – addresses hex FF00 to FF04 are not RAM, they hold a special memory control chip called the MMU (memory management unit).

Bank 1, for example, is almost entirely the RAM where Basic’s variables, arrays and strings are stored (RAM 1). But there’s a little bit of bank 0 still in there, at addresses 2 to 1023; and the MMU is still present at FF00 to FF04. In fact, these items will be there in all “normal” configurations.

Banks 0 and 1, then, are pure RAM, random access memory. You can store things there, and you can read the contents of these addresses. But you’d have trouble running most machine language programs in one of these banks (don’t let terminology throw you: I mean, “in one of these configurations”). You have no input/output paths available from these configurations, and you don’t have the built-in operating system (the ‘Kernal ROM’) to help the program do its job. In most cases, you’d find bank 15 (hex F) to be much more useful for running a program.

Excuse the hexadecimal numbers, but serious architecture students will want to see them that way. Bank 15 (F) has RAM 0 from address 2

to \$3FFF; above that is the ROM that holds the Basic logic, from 4000 to BFFF; above that is the Kernal operating system, in two chunks from C000 to CFFF and E000 to FFFF; and finally, the block from D000 to DFFF is used for the Input/Output (I/O) chips. If you need to use the character generator, Bank 14 (E) has the same architecture except that the block from D000 to DFFF contains the character set instead of I/O.

When you give a BANK command, nothing happens; the number you supply is stored (at address \$03D5). It won’t be used until you give a command which needs this number: POKE, PEEK, SYS, WAIT and some of the DOS commands such as BLOAD and BSAVE. Even then, the computer will only set up the configuration for a fleeting moment while it transfers material to or from the selected bank.

Roll Your Own

So you have only banks 0, 1, 14 and 15 for your work. No problem for a Basic programmer who might occasionally PEEK and POKE. But for the serious machine language programmer, it’s somewhat limiting. To keep the Kernal and I/O, the programmer is forced to select BANK 15; and that limits the program to RAM in the area below \$4000 (decimal 16384). This could be somewhat restricting, especially when a high-resolution screen might reside in the same area.

There’s hope. In fact, there are sixteen architectures that the ML programmer can use. Only four of them have BANK numbers, but the others can be reached by storing a value at \$FF00.

Table 1 shows all the practical combinations. Here’s a quick rundown on some of the most important:

00 – Storing this value in FF00 causes the C128 to take up its “normal” BANK 15 configuration. Use this before returning to Basic.

3F and 7F – Storing \$3F into FF00 creates the BANK 0 architecture; storing \$7F creates BANK 1. Careful: you have no I/O or Kernal ROM. There’s a shortcut to these architectures: storing anything to FF01 creates Bank 0; storing anything to FF02 creates Bank 1.

0E and 4E – Storing \$0E into FF00 creates the RAM 0 for addresses up to BFFF; storing \$4E creates RAM 1 for this area. The Kernal and I/O take up their normal positions. This are the “ideal” configurations for serious machine language stuff: 0E for a program in RAM 0, and 4E for a program in RAM 1. Basic is removed, and you have lots of memory to play with.

0F and 4F – These are similar to 0E and 4E above, except that the character generator chip is at addresses \$D000 to DFFF instead of I/O.

Use one of these configurations (briefly) when you need to examine the pixels of the character generator; but don't call any input or output when you are set up this way.

02, 03, 42 and 43 – These are curious configurations that keep the upper half of Basic (from 8000 to BFFF). They would not be used much except by enthusiasts who wanted to get at the floating point math routines in that area.

Summary

You can arrange any of a number of custom architectures if you need to. The standard BANKS are of limited help; use them to get from Basic and then organize your own architecture with a POKE to FF00.

Table 1. The sixteen 'useful' architectures.

FF00 (Poke Value)	Addresses whose first hex digits are:					Bank Number	Store to
0123	4567	89AB	CEF	D			
00	RAM0	ROM	ROM	ROM	I/O	"BANK 15"	FF03
01	RAM0	ROM	ROM	ROM	CGEN	"BANK 14"	
02	RAM0	RAM0	ROM	ROM	I/O		
03	RAM0	RAM0	ROM	ROM	CGEN		
0E	RAM0	RAM0	RAM0	ROM	I/O		FF01
0F	RAM0	RAM0	RAM0	ROM	CGEN		
3E	RAM0	RAM0	RAM0	RAM0	I/O		
3F	RAM0	RAM0	RAM0	RAM0	RAM0	"BANK 0"	
40	RAM1	ROM	ROM	ROM	I/O		FF04
41	RAM1	ROM	ROM	ROM	CGEN		
42	RAM1	RAM1	ROM	ROM	I/O		
43	RAM1	RAM1	ROM	ROM	CGEN		
4E	RAM1	RAM1	RAM1	ROM	I/O		FF02
4F	RAM1	RAM1	RAM1	ROM	CGEN		
7E	RAM1	RAM1	RAM1	RAM1	I/O		
7F	RAM1	RAM1	RAM1	RAM1	RAM1	"BANK 1"	

Note that in all configurations, the first 1K of memory (addresses 0002 to 03FF) is always RAM0. Addresses 0 and 1 are internal to the processor chip.

An Architecture-Testing Program

You might like to try your hand at checking the type of architecture that results when specific values are poked into location \$FF00. Run this program, supply a value, and see what you get.

The "business end" is a machine language program which tries the architecture and peeks various locations, reporting what it finds. Such a program must be tucked into the first 1K of memory: that's the only place that is safe from architecture switches.

The specific locations examined by the program are (hex): 3000, 6000, B000, F000, and D020. A value of 0 is poked to these locations in RAM 0, a value of 1 in RAM 1. The ROM values are fixed, hopefully: 6000 contains 60, b000 contains 4C, and F000 contains 29. At D020, the

character generator contains 78, and we make sure that the video chip border colour is set to its normal value of FD.

The machine language program sets the requested value into FF00, and then tests the contents of the specific locations. A zero is taken to be RAM 0; a 1 to be RAM 1; other values are tested for a match to the known ROM values. If none of these are recognized, the numeric value is printed. Each location is tested five times; if the value is not constant for every read, it's likely "not there" and is shown as VARYING.

C128 Architester

```

10 data 120, 141, 0, 255
20 data 174, 0, 48, 142, 128, 2
30 data 174, 0, 96, 142, 129, 2
40 data 174, 0, 176, 142, 130, 2
50 data 174, 0, 240, 142, 131, 2
60 data 174, 32, 208, 142, 132, 2
70 data 169, 0, 141, 0, 255, 88, 96
80 for j = dec("250") to dec("278")
90 read x:t = t + x:poke j,x
100 next j
110 if t <> 4305 then stop
120 for j = 3 to 0 step -1
130 bank j
140 poke dec("3000"),j:a(0,0) = -1
150 poke dec("6000"),j:a(1,0) = dec("60")
160 poke dec("b000"),j:a(2,0) = dec("4c")
170 poke dec("f000"),j:a(3,0) = dec("29")
180 poke dec("d020"),j:a(4,0) = dec("78")
190 next j
200 bank 15
210 poke dec("d020"),253
220 a$(0) = "0400-3fff"
230 a$(1) = "4000-7fff"
240 a$(2) = "8000-bfff"
250 a$(3) = "c000-cfff/e000-ffff"
260 a$(4) = "d000-dfff"
270 input "value of $ff00 poke(hex) ";x$
280 x = dec(x$):if x > 255 goto 270
290 for t = 1 to 5
300 sys dec("0250"),x
310 for j = 0 to 4:a(j,t) = peek(j + dec("0280")):next j
320 next t
330 for j = 0 to 4:q = fre(1)
340 a = a(j,1):r$ = ""
350 for t = 2 to 5:if a <> a(j,t) then a = 444
360 next t
370 if a = 0 then r$ = "ram0"
380 if a = 1 then r$ = "ram1"
390 if a = 2 then r$ = "ram2"
400 if a = 3 then r$ = "ram3"
410 if a = a(j,0) then r$ = "rom":if j = 4 then r$ = "cgen"
420 if j = 4 and a = 253 then r$ = "i/o"
430 if a = 120 then r$ = "cgen"
440 if a = 444 then r$ = "varies"
450 if r$ = "" then r$ = str$(a)
460 print a$(j); " - ";
470 print r$
480 next j

```


Getting The C128's CP/M+ In Gear

Clifton Karnes
Greensboro, NC

After stating we felt there was not enough demand for more CP/M info than is already available, we were deluged with letters. Several of the responses explained it was just the contrary - that what little CP/M info is around, is hard to find. So here is the first of what we hope will be more articles on C128 CP/M+. - EIC.

One of the nicest things about the new C128 and 1571 disk drive is that they have a CP/M mode that can read real CP/M disks. The system as supplied has some excellent features but unfortunately it is incomplete. There is, however, a solution.

In this article I will discuss how to get the C128's CP/M+ system up to par, how to begin tapping the huge source of public domain software, and describe some language implementations (both commercial and public domain) that I've tried on the C128 in CP/M mode.

Where's the Assembler?

The first thing you'll notice about the CP/M+ disks is that they contain no Assembler (MAC), debugger (SID), or any of the other utilities and source files that are supposed to come with CP/M+. This problem is easily solved. Just send in the card for the "DRI Special Offer" (and \$19.95). Commodore will send you the missing utilities and a huge manual.

Where's the I/O?

The next thing you'll notice about the CP/M+ is that, besides the console and disk, all the serial I/O routines are null. This means that the User's Port is dead. If your printer uses this port or you have a modem you would like to use, you are out of luck. But don't despair.

Where's the Standard ASCII?

The next question that may arise regards ASCII. CP/M uses standard ASCII and the 128 implies it does (see the SETKEY utility). This is true in part. The characters sent to the screen are standard representations (characters unusual to PetSCII are formed with the CTRL key plus the key that most nearly resembles the character eg. CTRL [and CTRL] for left and right curly brackets, CTRL / for backslash). But the codes sent to the printer are PetSCII and there's no way to change that. In other words, if you've got a flexible printer like the Star SG-10 and an interface, you can't get out of emulation mode to use any of the printer's extra features or for that matter its standard characters that aren't part of PetSCII. There's hope.

Commodore, CompuServe and Irv Hoff to the Rescue!

CBM Engineering (in the guise of Von Ertwine) has been working on these problems and there is a new approved CP/M+ operating system available free to all on CompuServe. This new operating system enables the serial I/O so your User's Port is undead. In

addition to the new operating system, there's a new utility called CONF that allows you to configure your system using an ASCII printer, dual disk drives, define baud rate, screen and cursor colors, key feel, and much more. If this weren't enough there is even a modem program for the 128 available on CompuServe: IMP by Irv Hoff. IMP is the latest CP/M modem program in the honorable line that began with MODEM7. This modem program is excellent and opens up the world to CP/Mers.

How can you get this stuff? First you must be a member of CompuServe. If you're not then this is a good time to join. You'll need VIDTEX 4.0C to start downloading. All of this material is in DL3 of CPM-IG. CPM-IG (the CP/M Special Interest Group on CompuServe) has started DL3 as a Data Library specifically for C128 CP/Mers. Nice.

The thing to do first is download C128.IRV. This file explains which other files are needed and how to get them. You'll need NEWSYS.COM (this creates a new CP/M+ operating system), IMP-C8.BIN and IMP.DOC (this is the modem program and its documentation), I2C8-1.ASM (this is an overlay to let you customize the modem program), CONF.COM and CONF.HLP (these allow you to set system parameters and tell you how), C1571.COM (this nearly doubles the write speed of the 1571 in CP/M mode). In addition there are two files to help you with the downloading process: BIBOOT.IMG (for single drive users) and 64CONV (for users with two drives). Even at 300 baud none of these files are long enough to be very expensive to download. I recommend that if you're not a member of CompuServe you join, but if for one reason or another the way of getting this software I've described isn't appealing, then you can send me a formatted CP/M+ disk, and SASE and \$3.00 and I'll copy the files for you.

Free Software

Now that you've got your system tuned up you'll want to get some free software. The best place to get started is to look into two books on the subject: Free Software by Robert A. Froelich (New York: Crown Publishers, 1984) and How to Get Free Software by Alfred Glossbrenner (New York: St. Martin's Press, 1984). Both of these works give excellent introductions to obtaining free software.

There are two basic ways to get public domain software: download it or buy the disks. You can download from a commercial database, like CompuServe or from a bulletin board. You don't actually buy public domain software (or shouldn't) but most user's groups charge a donation for equipment wear-and-tear, etc. and there are copy services that copy public domain programs for profit (you're paying

for their service – not the software). Which procedure is more economical? That depends on your situation. If you've got a 1200 bps modem and a local bulletin board or if the files you're interested in are fairly short then downloading is the way to go. If, however, you're not in this situation, it can become very expensive to download programs with all the relevant files. The costs vary with buying the disks themselves from something like a dollar a disk for local user's groups (you'll usually have to join the group too, which will be around \$10 - \$25) to \$15 and up for copy services.

There are two principal national sources of CP/M public domain software on disks: CP/M User's Group (CPMUG) and the Special Interest Group for Microcomputers (SIG/M). These groups both have extensive libraries. These books discuss both these sources at length. As for the formats that will work on your 128: Kaypro 2, 4, IBM CP/M 86 and Osborne Double-Density all work fine. That covers a lot of territory. Most of the public domain software out there is for CP/M 2.2, but I haven't found any incompatibilities yet with the CP/M 3.0 on the 128.

Programming Languages

The main reason many of you are interested in CP/M is the programming languages available. Many languages are even in the public domain. The most famous of these is perhaps Small-C. What follows is an annotated listing of commercial programming languages and editors I've tried that are low in price and that work on the 128, followed by some public domain packages available from the national user's groups mentioned above.

The Beginning

Although this is the end of our smorgasbord of information on C128 CP/M+, I hope it will be the beginning for you. Find a source and start checking the stuff out. Maybe the Transactor will even start giving a page each issue to C128 CP/M+ developments.

Clifton Karnes
2519 Overbrook Dr.
Greensboro, NC 27408
(919) 373-7892

Addresses of Software Sources Mentioned

Mix Software
2116 E. Arapaho
Suite 363
Richardson, TX 75081
(214) 783-6001

SIG/M Main Office (write to them to find your nearest SIG/M representative)
Box 97
Iselin, NJ 08830

Ellis Computing
3917 Noriega St.
San Francisco, CA 94122
(415) 753-0186

CPMUG
1651 Third Ave.
New York, NY 10128

Software Toolworks
14478 Glorietta Dr.
Sherman Oaks, CA 91423
(818) 986-4885

C User's Group (CUG)
Box 97
McPherson, KS 67460
(216) 241-1065

Language	Description	Price	Supplier
MIX C	Full K&R C compiler with UNIX functions, 400-page manual	\$39.95	MIX Software
Nevada FORTRAN	Fortran IV with '77 extensions	\$49.95	Ellis Computing
Nevada COBOL	ANSI COBOL '74 with level II features	\$49.95	Ellis Computing
LISP/80	InterLISP dialect	\$39.95	Software Toolworks
MIX Edit	Full screen / Split screen, programmable	\$29.95	Mix Software
Nevada Edit	Full screen	\$49.95	Ellis Computing
(I received excellent service from all of commercial sources listed above). In addition to these commercial packages there are several languages available in the public domain. These include:			
EBASIC	Gordon Eubank's Master's thesis and a forerunner of the widely used CBASIC	Free	CPMUG Volume 30
Small-C	'C' programming language by Ron Cain, with only int and char data types but widely used. Comes with source code.	Free	C User's Group
XLISP	Experimental Lisp by David Betz. Comes with source code in C. Soon to be upgraded to a subset of Common Lisp.	Free	SIG/M Volume 118
FORTH-83	Forth-83 implementation version 2.0	Free	SIG/M Volume 204
E-Prolog	A small Prolog implementation. Comes with ASM source and a VALGOL compiler written in Prolog.	Free	SIG/M Volume 242
JRT Pascal	Full Pascal implementation	Free	CPMUG Volume 82
This is just a list of the more popular languages available – there are others. And lots of other software including: assemblers, text editors, disk utilities (there are tons of these – one of the best is SWEEP in its latest version), and games. As the two books mentioned above show there is no such thing as completely free software, the price you pay for public domain programs (either in downloading time or to user's group or copy service) is usually a very small fraction of the value of work. Also, most of the above-mentioned public domain works can be downloaded from any number of sources if you choose that route.			

C128 RAM Disk

Noel Nyman, Seattle, WA

Add A 16K RAM Disk To Your C-128 With No Additional Hardware!

A RAM disk is a chunk of random access (read/write) memory that acts like a disk drive. LOAD and SAVE work with it. It cannot be reached by store, PEEK or POKE. RAM disk is external hardware, a circuit board with chips of various sorts. If it has enough memory to be practical it is physically large and expensive.

The advantage of RAM disk is speed. Files can be located and LOADED rapidly. Some database users find RAM disks worthwhile. But a 170K RAM disk costs about what you would pay for a 1541, and the memory goes away when the power is turned off.

If you own a C-128 you can try RAM disk with no additional outlay for hardware. Every C-128 has 16K of RAM that is not part of the regular memory map - the eighty column video RAM. Although a 16K RAM disk is small by commercial standards, it will hold 62 blocks of Basic programs or one-fourth of the Basic variable memory. A new Basic program, or a whole new set of variable values, can be brought into memory in about two seconds.

To understand how our RAM disk will work, you must know a little about the eighty column display system. We only have access to the 8563 eighty column video chip and its RAM through two addresses or "ports". One of these ports, located at Bank 15, address 54784 (\$D600 in hexadecimal), is used to select a register in the 8563. The other port, located at 54785 (\$D601 hex), is used to read from or write to the selected register. The video chip uses the register data to make changes on the screen. The buzzword used to describe this situation is "pipelined architecture".

There are 37 registers in the video chip. Some of them are high/low address vectors (pointers) to the video RAM. Others change eighty column screen functions by passing numbers or setting and clearing flags. We'll only be concerned with three of the registers.

Registers 18 and 19 hold the vector to an address in video RAM. The vector is stored in HIGH/LOW order. Machine language programmers are used to two byte addresses being stored in the opposite sequence, so it's important to note the difference.

Register 31 is the CPU Data register. The value at the address pointed to by registers 18/19 is available in register 31. If we access register 31 and store a number at the data port (\$D601), it will be placed at the video RAM address pointed to by registers 18/19. The vector at 18/19 is then incremented automatically.

When we store a new register value at address \$D600, the video chip is probably busy updating the eighty column screen. We have to wait until the chip is ready to look at our data, or we get erratic results. Bit #7 of address \$D600 is held low when the video chip is busy, and goes high when it is ready to accept new data. The ROM routine below is used by the 8502 processor to check bit #7.

```
FCDCA A2 1F LDX #$1F
CDCC 8E 00 D6 STX $D600
CDCF 2C 00 D6 BIT $D600
CDD2 10 FB BPL $CDCF
CDD4 8D 01 D6 STA $D601
CDD7 60 RTS
```

The video register to be accessed is stored in the X register of the 8502 and the routine is entered at address \$CDCC. The value in X is stored at \$D600. Then the BIT command checks for bit #7 to go high. Until it does, the BPL command will branch back to the BIT instruction. Once bit #7 goes high, the video chip is ready and the new data is stored at the data port, \$D601.

If we want to store data in register 31, the routine is entered at address \$CDCA which stores 31 (\$1F) in X for us. There is a complementary routine starting at \$CDD8 that reads a video chip register.

The following programs will store data to the 16K video RAM and retrieve it for later use. You must use the 40 column screen with them, since any printing done to the 80 column area will garble the data you've placed there. Before RUNNING the programs, switch to 40 column mode. If you have an 80 column monitor available, clear the 80 column screen and type the following:

POKE 54784,25; POKE 54785,128

Storing 128 in register 25 puts the video chip in Hi-Res or bit mapped mode. The two sets of vertical bars at the top of the screen are the text (CHR\$(32) on a cleared screen) and attribute (color ram) screens. The horizontal bars below are remnants of the RAM test done on power-up. The five columns at the bottom are the character sets data. The blank spaces are there for additional character information for a double wide character mode.

By switching to bit mapped mode, you'll be able to see the data and programs being SAVED to RAM disk.

Listing #1 is a Basic loader for a routine designed to copy portions of memory from any Bank to RAM disk. Enter the loader, RUN it, then SAVE the resulting machine language program by typing:

BSAVE "MEMORY DRAM", B0, P3072 TO P3184

To use the routine, set-up the beginning and ending addresses of memory using these commands:

SYS 3072,lb,hb

SYS 3077,le,he

Where:

lb = low byte of address of beginning of memory
hb = high byte of address of beginning of memory

le = low byte of address (+1) of end of memory
he = high byte of address (+1) of end of memory

For example, if you want to save all the variables (except dynamic string data) created by a Basic program, type:

SYS 3072, PEEK(47), PEEK(48)

SYS 3077, PEEK(51), PEEK(52)

Be sure that no more than 16K of memory is involved. Then type:

SYS 3082,0,0,1

The first two numbers after the SYS address are the low/high vector to the location in video RAM where the memory will be stored. The third number is the Bank number of the memory to be copied. For variables, this would be Bank 1.

After storing to RAM disk, type:

```
PRINT PEEK(251), PEEK(252)
```

This will give you the low/high vector of the next available location in video RAM. You can store several blocks of memory and retrieve them independently by keeping track of their video RAM starting addresses.

To get the data back, set up the starting and ending addresses as above. Then SYS to the routine using the same video ram address vectors. Add "128" to the Bank number to signal the routine to retrieve from the video RAM rather than copy to it. If retrieving variables for use with Basic, you should also POKE the appropriate values in locations 47/48 and 51/52.

If you want to use the eighty column text screen, you can still have access to 4K of disk RAM. The area between video RAM addresses 4049 and 8191 is unused in text mode. If you store more than 4K in this area, you'll overwrite the character set data.

Listing #2 is a Basic loader for a routine to SAVE and LOAD Basic programs. As before, enter the program, RUN it, and type the following to SAVE the machine code:

```
BSAVE "BASIC DRAM", B0, P2956 TO P3573
```

To activate the routine enter

```
SYS 2956.
```

BASIC DRAM Adds 3 Commands to the C-128.

MSAVE SAVes the Basic program in memory to RAM disk, assigns it a number, and shows the amount of memory remaining in RAM disk.

MLOAD LOADs a program from RAM disk to the current Basic memory space in Bank 0. The command must be followed immediately, no spaces, with the number (0-9) of a program already MSAVED.

MSCRATCH Asks for a starting program number and "scratches" that program and all programs with higher numbers from the RAM disk.

MLOAD can be followed by a colon and other commands in direct mode. For example:

```
MLOAD2 : RUN
```

will place program #2 from RAM disk into memory and RUN it.

The C-128 has two 256 byte pages permanently designated for RS-232 use that sit below the Basic program area. These are destined to become popular "safe" locations for machine code. The MEMORY DRAM code is located in the RS-232 input buffer. The BASIC DRAM program is longer and uses both buffers and the top of the tape buffer as well. If you RUN a Basic program that uses any of these buffers, the computer will probably "crash".

You can also use the video RAM from C-64 mode. The eighty column screen will be accessible, provided that you've used the command "GO64" after first booting in C-128 mode. Listings #3 and #4 are the C-64 mode versions of the MEMORY and BASIC DRAM programs. They are relocated to start at 51200 (\$C800) in the C-64 memory map. This is half way between the popular 49152 (\$C000) location used by many machine language routines and 52224 (\$CC00) used by the DOS 5.1 wedge.

To use the C-64 MEMORY DRAM, enter:

```
SYS 51200,lb,hb
```

```
SYS 51212,le,he
```

```
SYS 51224,0,0,0
```

The three zeroes after the last SYS represent any low/high byte address in video RAM and the flag to store or retrieve memory. Since there are no Banks in C-64 mode, use a zero to store and 128 to recover.

For C-64 BASIC DRAM, SYS 51200 to initialize the program. The same three commands are added and follow the same rules as C-128 mode, except that additional commands cannot be used on the same line as MLOAD.

To disable BASIC DRAM in either mode, use the reset switch near the on-off switch or manually change the ERROR vector at \$0300/\$0301 to its default value. The BASIC DRAM (for both modes) is a compromise between features and length. It will give you a "DISK FULL" error if you try to SAVE more than ten programs. But it doesn't check for actual memory left in video RAM. If you SAVE something too large, the address registers will merrily "roll over" to zero and store on top of data you've already placed there.

Assigning numbers to the programs is another compromise. It would have been best to intercept the LOAD and SAVE routines, assign an unused device number to the RAM disk, use file names, etc. This would have required a lot more code, too much to type in from a magazine listing.

Possibly the most significant compromise was made to allow the RUN-STOP/RESTORE key combination to halt Basic programs. The C-128 RESTORE routine clears both screens when executed. Since clearing a RAM disk isn't what we had in mind, the NMI vector is relocated to point to an abbreviated routine that leaves the RAM disk alone. The normal RESTORE resets several pointers, NMI among them. Since we can't have that either, the pointer routine was also eliminated. RESTORE uses several JSR calls to ROM routines. To leave out only small portions of these routines, we would have to put the balance of them in our program, and you would have to type them in. Instead, we've left out several of the JSR calls, and kept the minimum to get Basic to work properly. If you have a favorite program that uses any machine code, test it thoroughly when using it with BASIC DRAM.

These problems don't plague the C-64 version. There is no eighty column screen to clear on a C-64, so RESTORE doesn't have that function. We can leave the normal NMI routines intact.

We did have to add some code to the C-64 version, however. The SYS command in the C-128 looks for values separated by commas following the SYS address. The first three will be transferred to the A, X, and Y registers of the 8502. This makes passing values to ML short and sweet. (A fourth value will be placed in the Status Register, but beware of that! The value placed there will affect all the flags, including decimal mode. The processor will also set bit #5, the unused flag, even if your passed value left in clear.)

The C-64 doesn't have this feature, and to keep the commands the same, the code must be added. We also needed to add the routines to access the register and data ports for the video chip.

Both BASIC DRAM's use the error wedge technique described by Brian Munshaw in Transactor 5-6 ("A New Wedge for the Commodore 64"). The three added commands cause a "syntax error". Our program intercepts all error messages and passes on any except syntax errors. These are examined for the use of an illegal character in front of a LOAD, SAVE, or SCRATCH token. Since any illegal character will work, XSAVE will have the same effect as MSAVE.

The C-64 doesn't tokenize the word "scratch", so some additional code is required. To eliminate excess typing, we've decoded only the "sc" portion. MSCREAM will work as well for MSCRATCH.

We hope that you enjoy experimenting with RAM disk and find it useful. For example, you could MSAVE several programs such as Disk Doctor, Directory Reorganizer, Two Column Directory Printer, etc. prior to a heavy disk reorganization session. Then a simple MLOAD# will quickly bring in each program as you need it. This would be a real advantage to anyone using a 1541/C-128 combination.

The Merlin source code for the DRAM programs will be found on The Transactor Disk for this issue. If you'd prefer hard copy of the source code, mail \$2 (either Canadian or US) and a large addressed envelope to:

Noel Nyman
Geoduck Developmental System
PO Box 58587
Seattle, WA 98188

Listing One

```
LL 1000 rem save "0:mem dram.ldr",8
GG 1010 :
CH 1020 for j=3072 to 3183: read x: poke j,x:
    ch = ch + x: next
LJ 1030 if ch<>19056 then print "checksum error"
EI 1040 :
EE 1050 data 133, 251, 134, 252, 96, 133, 253, 134
PC 1060 data 254, 96, 133, 200, 134, 201, 132, 250
KM 1070 data 72, 138, 162, 18, 32, 204, 205, 232
HK 1080 data 104, 32, 204, 205, 160, 0, 165, 250
ML 1090 data 48, 42, 166, 250, 169, 251, 32, 116
CP 1100 data 255, 32, 202, 205, 230, 251, 208, 2
MJ 1110 data 230, 252, 165, 251, 197, 253, 208, 234
KI 1120 data 165, 252, 197, 254, 208, 228, 162, 18
HL 1130 data 32, 218, 205, 133, 252, 232, 32, 218
BP 1140 data 205, 133, 251, 96, 41, 15, 133, 250
LA 1150 data 169, 251, 141, 185, 2, 32, 216, 205
OL 1160 data 166, 250, 32, 119, 255, 230, 251, 208
KH 1170 data 2, 230, 252, 165, 251, 197, 253, 208
NM 1180 data 236, 165, 252, 197, 254, 208, 230, 96
```

Listing Two

```
JJ 2000 rem save "0:bas dram.ldr",8
OE 2010 :
LH 2020 for j=2956 to 3571: read x: poke j,x:
    ch = ch + x: next
JJ 2030 if ch<>69893 then print "checksum error"
MG 2040 :
```

```
ID 2050 data 120, 173, 0, 3, 201, 63, 208, 29
GB 2060 data 141, 179, 11, 173, 1, 3, 141, 180
AI 2070 data 11, 169, 181, 141, 0, 3, 169, 11
EM 2080 data 141, 1, 3, 169, 195, 141, 24, 3
LI 2090 data 169, 13, 141, 25, 3, 88, 96, 0
FF 2100 data 0, 224, 11, 240, 3, 108, 179, 11
MC 2110 data 201, 147, 208, 3, 76, 208, 12, 201
GL 2120 data 148, 240, 7, 201, 242, 208, 238, 76
AG 2130 data 60, 13, 169, 0, 141, 0, 255, 173
NE 2140 data 223, 13, 201, 10, 144, 28, 32, 125
MC 2150 data 255, 13, 18, 82, 65, 77, 32, 68
AN 2160 data 73, 83, 75, 32, 70, 85, 76, 76
IK 2170 data 27, 81, 141, 0, 32, 142, 201, 76
JH 2180 data 182, 12, 10, 168, 133, 200, 185, 224
EA 2190 data 13, 162, 18, 32, 204, 205, 185, 225
JI 2200 data 13, 232, 32, 204, 205, 56, 173, 16
DF 2210 data 18, 133, 253, 229, 45, 32, 202, 205
MN 2220 data 173, 17, 18, 133, 254, 229, 46, 32
HA 2230 data 202, 205, 165, 45, 133, 251, 165, 46
JN 2240 data 133, 252, 160, 0, 162, 0, 169, 251
FM 2250 data 32, 116, 255, 32, 202, 205, 230, 251
IL 2260 data 208, 2, 230, 252, 165, 251, 197, 253
MB 2270 data 208, 234, 165, 252, 197, 254, 208, 228
LP 2280 data 164, 200, 200, 200, 162, 18, 32, 218
NA 2290 data 205, 133, 252, 153, 224, 13, 232, 32
AB 2300 data 218, 205, 133, 251, 153, 225, 13, 32
FE 2310 data 125, 255, 141, 83, 65, 86, 69, 68
CG 2320 data 32, 80, 82, 79, 71, 82, 65, 77
EK 2330 data 32, 0, 169, 0, 174, 223, 13, 32
PE 2340 data 187, 12, 32, 125, 255, 27, 81, 141
BI 2350 data 32, 32, 0, 56, 169, 128, 229, 251
PP 2360 data 170, 169, 62, 229, 252, 32, 187, 12
OM 2370 data 32, 125, 255, 32, 66, 89, 84, 69
EM 2380 data 83, 32, 82, 69, 77, 65, 73, 78
CM 2390 data 73, 78, 71, 32, 73, 78, 32, 82
HM 2400 data 65, 77, 32, 68, 73, 83, 75, 32
HI 2410 data 27, 81, 141, 0, 238, 223, 13, 32
AF 2420 data 142, 201, 162, 128, 108, 179, 11, 160
IL 2430 data 0, 132, 98, 133, 97, 134, 96, 32
DL 2440 data 7, 186, 169, 0, 162, 8, 160, 3
MB 2450 data 32, 93, 186, 96, 32, 128, 3, 41
AM 2460 data 15, 205, 223, 13, 144, 3, 76, 158
FL 2470 data 13, 10, 168, 169, 0, 141, 0, 255
KE 2480 data 165, 200, 185, 224, 13, 162, 18, 32
JJ 2490 data 204, 205, 133, 252, 232, 185, 225, 13
DM 2500 data 32, 204, 205, 32, 216, 205, 133, 253
MH 2510 data 32, 216, 205, 133, 254, 24, 165, 45
CM 2520 data 133, 251, 101, 253, 133, 253, 141, 16
DP 2530 data 18, 165, 46, 133, 252, 101, 254, 133
GB 2540 data 254, 169, 251, 141, 185, 2, 160, 0
BD 2550 data 32, 216, 205, 162, 0, 32, 119, 255
EO 2560 data 230, 251, 208, 2, 230, 252, 165, 251
BH 2570 data 197, 253, 208, 236, 165, 252, 197, 254
MD 2580 data 208, 230, 32, 79, 79, 76, 162, 82
HB 2590 data 169, 0, 141, 0, 255, 32, 125, 255
AA 2600 data 141, 83, 67, 82, 65, 84, 67, 72
EJ 2610 data 32, 83, 84, 65, 82, 84, 73, 78
HJ 2620 data 71, 32, 87, 73, 84, 72, 32, 80
NM 2630 data 71, 77, 32, 78, 85, 77, 66, 69
EM 2640 data 82, 32, 63, 32, 0, 32, 228, 255
LI 2650 data 201, 0, 240, 249, 32, 210, 255, 201
PG 2660 data 48, 144, 39, 201, 58, 176, 35, 41
NO 2670 data 15, 205, 223, 13, 176, 28, 141, 223
DN 2680 data 13, 133, 200, 230, 200, 6, 200, 164
```


KM	2690 data 200, 169, 0, 153, 224, 13, 153, 225
FP	2700 data 13, 200, 200, 192, 20, 208, 244, 76
LK	2710 data 182, 12, 32, 125, 255, 141, 18, 73
DD	2720 data 78, 86, 65, 76, 73, 68, 32, 80
AD	2730 data 82, 79, 71, 82, 65, 77, 32, 78
BI	2740 data 85, 77, 66, 69, 82, 27, 81, 141
KJ	2750 data 0, 32, 142, 201, 76, 182, 12, 216
EO	2760 data 169, 127, 141, 13, 221, 172, 13, 221
KL	2770 data 32, 61, 246, 32, 225, 255, 208, 8
JB	2780 data 169, 147, 32, 210, 255, 108, 0, 10
NO	2790 data 76, 51, 255, 0, 0, 0, 0, 0
OL	2800 data 0, 0, 0, 0, 0, 0, 0, 0
IM	2810 data 0, 0, 0, 0, 0, 0, 0, 0

Listing Three

ND	1000 rem save "0:64mem dram.ldr", 8
GG	1010 :
HG	1020 for j=51200 to 51374: read x: poke j,x: ch=ch+x: next
CJ	1030 if ch<>26408 then print "checksum error"
EI	1040 :
HD	1050 data 32, 152, 200, 165, 170, 133, 251, 165
KE	1060 data 171, 133, 252, 96, 32, 152, 200, 165
NF	1070 data 170, 133, 253, 165, 171, 133, 254, 96
II	1080 data 32, 141, 200, 165, 169, 133, 167, 165
IG	1090 data 170, 133, 168, 162, 18, 32, 115, 200
CO	1100 data 232, 165, 167, 32, 115, 200, 160, 0
IP	1110 data 165, 171, 48, 37, 177, 251, 32, 113
DE	1120 data 200, 230, 251, 208, 2, 230, 252, 165
BN	1130 data 251, 197, 253, 208, 239, 165, 252, 197
FJ	1140 data 254, 208, 233, 162, 18, 32, 129, 200
NH	1150 data 133, 252, 232, 32, 129, 200, 133, 251
DH	1160 data 96, 32, 127, 200, 145, 251, 230, 251
GH	1170 data 208, 2, 230, 252, 165, 251, 197, 253
ON	1180 data 208, 239, 165, 252, 197, 254, 208, 233
BE	1190 data 96, 162, 31, 142, 0, 214, 44, 0
OO	1200 data 214, 16, 251, 141, 1, 214, 96, 162
NB	1210 data 31, 142, 0, 214, 44, 0, 214, 16
BP	1220 data 251, 173, 1, 214, 96, 32, 253, 174
BB	1230 data 32, 158, 173, 32, 170, 177, 132, 169
GG	1240 data 32, 253, 174, 32, 158, 173, 32, 170
AE	1250 data 177, 132, 170, 32, 253, 174, 32, 158
FH	1260 data 173, 32, 170, 177, 132, 171, 96

Listing Four

BA	2000 rem save "0:64bas dram.ldr", 8
OE	2010 :
MF	2020 for j=51200 to 51777: read x: poke j,x: ch=ch+x: next
JI	2030 if ch<>67582 then print "checksum error"
MG	2040 :
MN	2050 data 120, 173, 0, 3, 201, 139, 208, 19
FE	2060 data 141, 29, 200, 173, 1, 3, 141, 30
BB	2070 data 200, 169, 31, 141, 0, 3, 169, 200
CF	2080 data 141, 1, 3, 88, 96, 0, 0, 224
BP	2090 data 11, 240, 3, 108, 29, 200, 32, 121
IA	2100 data 0, 201, 147, 208, 3, 76, 234, 200
PL	2110 data 201, 148, 240, 21, 169, 1, 133, 122
FC	2120 data 32, 121, 0, 201, 83, 208, 228, 32
MB	2130 data 115, 0, 201, 67, 208, 221, 76, 72
LL	2140 data 201, 173, 45, 202, 201, 10, 144, 10
LO	2150 data 169, 170, 162, 201, 32, 30, 171, 76

FO	2160 data 229, 200, 10, 168, 133, 167, 185, 46
MO	2170 data 202, 162, 18, 32, 144, 201, 185, 47
HA	2180 data 202, 232, 32, 144, 201, 56, 165, 45
CJ	2190 data 133, 253, 229, 43, 32, 142, 201, 165
DE	2200 data 46, 133, 254, 229, 44, 32, 142, 201
EO	2210 data 165, 43, 133, 251, 165, 44, 133, 252
MO	2220 data 160, 0, 177, 251, 32, 142, 201, 230
IK	2230 data 251, 208, 2, 230, 252, 165, 251, 197
KA	2240 data 253, 208, 239, 165, 252, 197, 254, 208
KP	2250 data 233, 164, 167, 200, 200, 162, 18, 32
NM	2260 data 158, 201, 133, 252, 153, 46, 202, 232
GC	2270 data 32, 158, 201, 133, 251, 153, 47, 202
LB	2280 data 169, 187, 160, 201, 32, 30, 171, 169
GA	2290 data 0, 174, 45, 202, 32, 205, 189, 169
LF	2300 data 203, 160, 201, 32, 30, 171, 56, 169
HD	2310 data 128, 229, 251, 170, 169, 62, 229, 252
BJ	2320 data 32, 205, 189, 169, 207, 160, 201, 32
ND	2330 data 30, 171, 238, 45, 202, 162, 128, 108
HB	2340 data 29, 200, 32, 115, 0, 176, 246, 41
GC	2350 data 15, 205, 45, 202, 144, 3, 76, 132
MM	2360 data 201, 10, 168, 185, 46, 202, 162, 18
KN	2370 data 32, 144, 201, 232, 185, 47, 202, 32
AI	2380 data 144, 201, 32, 156, 201, 133, 253, 32
HI	2390 data 156, 201, 133, 254, 24, 165, 43, 133
HG	2400 data 251, 101, 253, 133, 253, 133, 45, 165
MI	2410 data 44, 133, 252, 101, 254, 133, 254, 133
DK	2420 data 46, 160, 0, 32, 156, 201, 145, 251
CG	2430 data 230, 251, 208, 2, 230, 252, 165, 251
CP	2440 data 197, 253, 208, 239, 165, 252, 197, 254
AA	2450 data 208, 233, 32, 51, 165, 76, 229, 200
OA	2460 data 169, 238, 160, 201, 32, 30, 171, 32
DP	2470 data 228, 255, 201, 0, 240, 249, 32, 210
EF	2480 data 255, 201, 48, 144, 39, 201, 58, 176
JL	2490 data 35, 41, 15, 205, 45, 202, 176, 28
KJ	2500 data 141, 45, 202, 133, 167, 230, 167, 6
PF	2510 data 167, 164, 167, 169, 0, 153, 46, 202
LP	2520 data 153, 47, 202, 200, 200, 192, 20, 208
DE	2530 data 244, 76, 229, 200, 169, 19, 160, 202
JN	2540 data 32, 30, 171, 76, 229, 200, 162, 31
LO	2550 data 142, 0, 214, 44, 0, 214, 16, 251
CP	2560 data 141, 1, 214, 96, 162, 31, 142, 0
JC	2570 data 214, 44, 0, 214, 16, 251, 173, 1
FI	2580 data 214, 96, 141, 18, 82, 65, 77, 32
HI	2590 data 68, 73, 83, 75, 32, 70, 85, 76
OP	2600 data 76, 141, 0, 141, 83, 65, 86, 69
LK	2610 data 68, 32, 80, 82, 79, 71, 82, 65
BD	2620 data 77, 32, 0, 141, 32, 32, 0, 32
HM	2630 data 66, 89, 84, 69, 83, 32, 82, 69
JM	2640 data 77, 65, 73, 78, 73, 78, 71, 32
IN	2650 data 73, 78, 32, 82, 65, 77, 32, 68
BO	2660 data 73, 83, 75, 32, 141, 0, 141, 83
AP	2670 data 67, 82, 65, 84, 67, 72, 32, 83
HO	2680 data 84, 65, 82, 84, 73, 78, 71, 32
GO	2690 data 87, 73, 84, 72, 32, 80, 71, 77
BA	2700 data 32, 78, 85, 77, 66, 69, 82, 32
NO	2710 data 63, 32, 0, 141, 18, 73, 78, 86
DA	2720 data 65, 76, 73, 68, 32, 80, 82, 79
GB	2730 data 71, 82, 65, 77, 32, 78, 85, 77
BE	2740 data 66, 69, 82, 141, 0, 0, 0, 0
MI	2750 data 0, 0, 0, 0, 0, 0, 0, 0
GJ	2760 data 0, 0, 0, 0, 0, 0, 0, 0
IE	2770 data 0, 0

AmigaBasic Function Plot

Chris Zamara, Technical Editor

An Auto-Scaling Plotting Demo

This program will open a new window with all the standard gadgets and display the graph of a function within it, including the X and Y axes. The graph always fills the entire window, and will re-plot to a new size if you resize the window with its sizing gadget. The function to be plotted is defined in the program with a DEF FN statement, over a range of X values defined by the variables DOMAIN1 and DOMAIN2.

Before plotting the function, the program finds the highest and lowest values of the function so that it can scale to the size of the output window. The message "Scaling. . ." will be printed while this process takes place. After scaling, the function is plotted, taking up the entire height of the window, with the lines $X=0$ and $Y=0$ plotted in colour 2 (default colour black). You can move the output window around with the drag bars in the usual manner, and if you re-size the window, the program re-plots the function to fill the window at its new size. Since the function is only re-evaluated for each pixel in the width of the window, you'll find that the function plots faster when the window width is smaller.

When the output window is first opened by the program, it is sized so that a function is aspect-ratio corrected. That is, the X and Y co-ordinates are the same size on the screen, if not the same number of pixels. Thus, the function $Y=X$ will describe a true 45-degree angle. This, of course, can be changed by re-sizing the window, stretching the function in the X or Y direction.

The output window is opened and selected by the following line in the program:

```
WINDOW 2,title$(5,10)-(502,115),31 'new window
```

The above command will open a NEW window, leaving the standard BASIC window in place. The window is also auto-refreshed: moving it around won't mess up what's inside. That takes up a lot of memory, so if you only have 256K, you'll have to change it to WINDOW 1, replacing the BASIC

window. Or, you can just remove the line altogether, using the BASIC window with its original size and title.

To set up the function to be plotted, just change the DEF FN function definition as shown in the listing, and change the DOMAIN1 and DOMAIN2 variable assignments to define the start and end X values for which the function is evaluated. as listed, the program will plot the function $Y=\sin(X)$ from 0 to 2π , which is good for demonstration purposes but a bit boring. Several other functions appear as comments, along with recommended domain parameters. Take out the comment character "'" (apostrophe) and comment out the "DEF FN $Y(X)=\sin(X)$ ", then set up the DOMAIN1 and DOMAIN2 variable assignments to try one of the listed functions.

The program uses many of AmigaBasic's advanced capabilities. It uses no line numbers or labels, using control structures to control program flow. The scaling and plotting are done by local procedures, which only affect the required variables and produce no side effects like a standard BASIC subroutine does.

The only bug I know about is that sometimes the function will re-plot twice after a window re-sizing operation. It probably occurs when the window is re-sized between checks for window width and height.

The method of providing the function to the program is obviously primitive. A more polished program could easily grow from the humble bit of code presented here today. Pull-down menus could be used to select functions and the domain of the functions. A good idea might be a kind of "function construction kit", pulling out individual terms of an equation and combining them to create the desired function. Another good idea might be to allow different functions to be plotted on different windows, or maybe on the same window. A fairly easy feature to add would be a magnify function: pick a start and end point on the graph, and re-plot the chosen section. You could also get it to plot pre-calculated data from DATA statements or a disk file.


```
' function plot from Transactor Magazine
' this program may be freely distributed
' Mar 86 - CZ
' Plots any function and scales
' to the size of the output window.
'
' Set the function using DEF FN below
' and set the range of X values
' with the 'domain1' and 'domain2' variables.
'
pi = 3.141592
```

```
'put your function below. . .
title$ = "y = sin(x)" 'output window title
DEF FN y(x) = SIN(x) 'use 0 to 2*pi for domain
```

```
' . . .or try one of these
' DEF FN y(x) = SIN(x) + COS(2*x) 'domain=(0, 2*pi)
' DEF FN y(x) = SIN(x) + 2*SIN(15*x) 'domain=(-pi, +pi)
' DEF FN y(x) = -5*x-2*x*x-3*x*x*x 'try (-10, +10)
' DEF FN y(x) = SQR(9-x*x) '(-3, +3)
```

```
'..set the domain of X values here
domain1 = 0 'x start
domain2 = 2*pi 'x end
```

```
'=====
' make new window to display graph
WINDOW 2,title$(5,10)-(502,115),31 'new window
' find highest and lowest y values for scaling
CALL scale(range1, range2)
```

```
prev.width=0: prev.height=0
WHILE 1 'continuous loop
    new.width = WINDOW(2)
    new.height = WINDOW(3)
    'plot graph if window is re-sized
    IF new.width<>prev.width OR new.height<> prev.height THEN
        CALL PlotGraph(range1, range2)
    END IF
    prev.width = new.width
    prev.height = new.height
WEND
```

```
SUB scale(range1, range2) STATIC
' find max. and min. y values of function
' from domain1 to domain2
```

```
SHARED domain1, domain2
SHARED FN y()
```

```
PRINT "Scaling. . ."
s = (domain2-domain1)/WINDOW(2)
range1 = FN y(domain1)
range2 = range1
```

```
FOR x = domain1 TO domain2 STEP s
    y = FN y(x)
    IF y < range1 THEN range1 = y
    IF y > range2 THEN range2 = y
NEXT x
END SUB
```

```
SUB PlotGraph(range1, range2) STATIC
' Plot Graph of function Y to scale
' of current output window
```

```
SHARED domain1, domain2
SHARED FN y()
```

```
window.width = WINDOW(2)-1
window.height = WINDOW(3)-1
X.scale = (domain2-domain1)/window.width
Y.scale = window.height/(range2-range1)
Y.zero = range2*Y.scale
X.zero = -domain1/X.scale
```

```
'draw axis: lines y=0 and x=0
CLS
LINE (0, Y.zero)-(window.width, Y.zero),2
LINE (X.zero, 0)-(X.zero, window.height),2
'plot first point
PSET (0, Y.zero-FN y(domain1)*Y.scale)
'now plot whole function
FOR x.pixel = 0 TO window.width
    x = x.pixel*X.scale + domain1
    y = FN y(x)
    y.pixel = Y.zero - y*Y.scale
    LINE -(x.pixel, y.pixel)
NEXT x.pixel
END SUB
```


Kernel Routines In The B128

Liz Deal
Malvern, PA

This is a list of 46 KERNEL routines in the B128. It is somewhat different from the list in the Protecto/CBM Guide. Most of the routines in the B128 are similar to the C64, but some call addresses have been changed, setup registers sometimes differ, and there is more impact on the registers than was the case with the C64. This list is also valid for the B256 models which have the same Kernel ROM as the B128. Some B256 machines (in Europe) may have a different Kernel ROM. They can be distinguished from the most recent version by the presence of code in the "patch area", \$ECB0-ECE8.

Making this list would not have been possible without help from Jim Butterfield in the form of memory maps and a superb disassembler.

Unless otherwise noted, long addresses are normally sent/returned in this order: A=bank#, Y=high byte, X=low byte of address.

Usually in zero page, it is kept in the lo-hi-bank order. Often a register points to the first of the three bytes.

A, X, Y are data registers. If unchecked, it means, positively, that the routine has no effect on the register. The "C" column refers to the carry flag. It is a rare subroutine that does not affect the C status. So to avoid ambiguous clutter, the only time C is checked off is when it means something. Much of the time in the I/O routines C indicates an error, but ST does the job better – it may show an error while C is clear. ST=64 at the end of file; this is not indicated in the table below.

Jumbo Jump Table in Chronological Order – CBM names

ffb6	jmp \$fe9d	;txjump	transfer of execution jump	ffb7	jmp \$fb4a	;readst	read/set st
ffb6	jmp \$fbca	;vreset	power on/off vector reset	ffb8	jmp \$fb43	;setlfs	set files la,fa,sa
ff72	jmp \$fe33	;ipcgo	loop for ipc system	ffbd	jmp \$fb34	;setnam	set file name length and adrs.
ff75	jmp \$e022	;funkey	function key vector	ffc0	jmp (\$306) \$f6bf	;open	open logical file
ff78	jmp \$fcab	;iprst	send ipc request	ffc3	jmp (\$308) \$f5ed	;close	close/abort logical file
ff7b	jmp \$f9fb	;ioinit	i/o initialization	ffc6	jmp (\$30a) \$f549	;chkin	connect input channel
ff7e	jmp \$e004	;cint	screen initialization	ffc9	jmp (\$30c) \$f5a3	;chkout	connect output channel
ff81	jmp \$f400	;alocat	allocation of memory	ffcc	jmp (\$30e) \$f6a6	;clrchn/restio	reset default i/o devices
ff84	jmp \$fba9	;vector	read/set i/o vectors	ffcf	jmp (\$310) \$f49c	;chrin/basin/input	input a byte from open ch.
ff87	jmp \$fba2	;restor	restore i/o vectors	ffd2	jmp (\$312) \$f4ee	;chrout/basout	output a byte to open ch.
ff8a	jmp \$f660	;lkupsa	match sa	ffd5	jmp (\$31a) \$f746	;load	load from file
ff8d	jmp \$f678	;lkupla	match la	ffd8	jmp (\$31c) \$f84c	;save	save to file
ff90	jmp \$fb5a	;setmsg	enable/disable os messages	ffde	jmp \$f90e	;settim	set TOD clock
ff93	jmp (\$324) \$f274	;second	send sa after listen	ffde	jmp \$f8e6	;rdtim	read TOD clock
ff96	jmp (\$326) \$f280	;talksa	send sa after talk	ffe1	jmp (\$314) \$f96b	;stop	check STOP key
ff99	jmp \$fb78	;memtop	set/read top of memory	ffe4	jmp (\$316) \$f43d	;getin	get byte from KB or channel
ff9c	jmp \$fb8d	;membot	set/read bottom of memory	ffe7	jmp (\$318) \$f67f	;clall	close or abort files
ff9f	jmp \$e013	;scnkey	scan keyboard	ffea	jmp \$f979	;udtim	last row KB scan
ffa2	jmp \$fb74	;settmo	set ieee timeout	ffed	jmp \$e010	;scrorg/screen	return screen size
ffa5	jmp (\$328) \$f30a	;acptr	handshake ieee byte in	fff0	jmp \$e019	;plot	read/set cursor position
ffa8	jmp (\$32a) \$f297	;ciout	handshake ieee byte out	fff3	jmp \$e01c	;iobase	return i/o base address
ffab	jmp (\$32c) \$f2ab	;untilk	send untalk to ieee				
ffae	jmp (\$32e) \$f2af	;unlsn	send unlisten to ieee	fff6	sta \$0:rts	;goodbye	goes to another bank
ffb1	jmp (\$330) \$f234	;listen	send listen to ieee	fff9	.byte 1		
ffb4	jmp (\$332) \$f230	;talk	send talk to ieee	fffa	Hardware vectors: nmi \$fb31, reset \$f997, irq \$fbd6.		

B128 Kernel Routines

CBM Label	Jump addr	Ind addr	Real code	Operation Details		IN A X Y C	MOD A X Y C	MOD ST
ACPTR	FFA5	328	f30a	Get byte from IEEE	out: C = 1 and ST = 2 if timeout	----	a--c	*
ALOCAT	FF81	---	f400	Allocate YX bytes relative to top of user memory	in: X = low Y = high out: C = 1 if failed (use MEMTOP)	-xy-	axy c	
CHKIN	FFC6	30a	f549	Open channel for input	in: X = logical file# out: C = 0 if keyboard or RS232 if IEEE, C = 1 if no file, no device	-x--	ax-c	
CHKOUT	FFC9	30c	f5a3	Open channel for output	(see CHKIN)	-x--	ax--	
CHRIN	FFCF	310	f49c	Input character	out: C = 0 if keyboard or IEEE (use ST) RS232 if STOPped turns C = 1	----	a--c	*
CHROUT	FFD2	312	f4ee	Output character	out: C = 0 if screen or IEEE (use ST) RS232 if STOPped turns C = 1	a----	---c	*
CINT	FF7E	---	e044	Initialize screen editor, top of user memory, function keys		----	axy-	
CIOUT	FFA8	32a	f297	Output byte to IEEE	out: C = 1 and ST = 1 if timeout	a----	---c	*
CLALL	FFE7	318	f67f	Close or abort all files	in: A = dev#, C = 0 aborts, does CLRCHN C = 1 closes until error, aborts files after first error	a--c	axy c	
CLOSE	FFC3	308	f5ed	Close one file	in: A = log. file#, C = 0 aborts file, C = 1 real close file	a--c	axy-	*
CLRCHN	FFCC	30e	f6a6	Restore default devices		----	ax--	
FUNKEY	FF75	---	e6f8	Print/Edit function key definitions	Print all dfns- in: Y = 0 Edit key- in: Y = key# A = zero pg ptr to length of dfn and long addr.	-y- a-y-	axy- axy-	
GETIN	FFE4	316	f43d	Get a byte	out: XY unchanged in RS232/IEEE C = 0 in keyboard, RS232, IEEE (use ST)	----	axy-	*
IOBASE	FFF3	---	e03a	Returns address of I/O devices	out: bank 15, X = low, Y = high addr	----	-xy-	
IOINIT	FF7B	---	f9fb	Initialize I/O and TOD clock		----	axy-	
IPCGO	FF72	---	fe33	Loop for other processor		----	axy-	
IPRQST	FF78	---	fcab	Send ipc request		----	axy-	
LISTEN	FFB1	330	f234	Make IEEE device listen	in: A = device#	a----	a----	*
LKUPLA	FF8D	---	f678	Lookup parameters for file#	in: A = log. file# out: A = log. file#, X = dev# Y = sec addr, C = 1 if no file LA	a----	axy c	
LKUPSA	FF8A	---	f660	Lookup parameters on known sec. adrs	in: Y = secondary address out: A = log. file#, X = dev# Y = sec addr, C = 1 if no file matches SA	--y-	axy c	
LOAD	FFD5	31a	f746	Load after call to SETLFS, SETNAM	in: A bit 7 = 0 to load, bit 7 = 1 to verify, bits 0-3 dest. bank# Y, X = destination addr hi, lo (X = Y = \$FF to load at header addr) out: A, Y, X = long addr. last byte in.	axy-	axy-	*
MEMBOT	FF9C	---	fb8d	Read/Set bottom of memory	set: C = 0; A, Y, X = long address read: C = 1; A, Y, X = long address	axy c ---c	--- axy-	
MEMTOP	FF99	---	fb78	Read/Set top of memory	in: C = 0; A, Y, X = long address out: C = 1; A, Y, X = long address	axy c ---c	--- axy-	
OPEN	FFC0	306	f6bf	Open a logical file	in: C = 0 for normal open C = 1 temp IEEE channel, no table.	---c	axy-	
PLOT	FFF0	---	e025	Read/Set cursor position	read- in: C = 1 out: X = row Y = column set- in: C = 0; X = row Y = column	---c -xy c	-xy- a---	
RDTIM	FFDE	---	f8e6	Read TOD clock in BCD (see book for bit assignments)		----	axy-	
READST	FFB7	---	fb4a	Read/Set ST	read- in: C = 1 set- in: C = 0 A = value to go to ST	---c a--c	a--- ---	
RESTOR	FF87	---	fb2	Restore system default vectors		----	axy-	
SAVE	FFD8	31c	f84c	Save any memory	in: X = zero pg ptr to long start addr Y = zero pg ptr to long end adrs out: AX Y are NOT final address	-xy-	axy-	*
SCNKEY	FF9F	---	e013	Scan keyboard		----	axy-	
SCREEN	FFED	---	e010	Return screen size X = columns Y = rows		----	-xy-	
SECOND	FF93	324	f274	Send secondary adrs after listen		a----	----	*
SETLFS	FFBA	---	fb43	Set file parameters	in: A = log. file#, X = dev#, Y = sec addr	axy-	----	
SETMSG	FF90	---	fb5a	Enable/Disable OS messages	in: A bit 7 = 1 KERNEL msgs on, bit 6 = 1 Control msgs on	a----	----	
SETNAM	FFBD	---	fb34	Set file name	in: A = length of file name, X = zero pg ptr to long name addr	ax--	a---	
SETTIM	FFDB	---	f90e	Set TOD clock using BCD values	in: bits assignments-see book	axy-	a---	
SETTMO	FFA2	---	fb74	Enable/Disable IEEE timeout	in: A bit 7 = 0 enable, bit 7 = 1 disable	a----	----	
STOP	FFE1	314	f96b	Check Stop key	out: Z = 0 if STOP not used; X unchanged Z = 1 if STOP used, X changed by call to CLRCHN	----	ax--	
TALK	FFB4	332	f230	Make IEEE device talk	in: A = device#	a----	a---	*
TLKSA	FF96	326	f280	Send secondary address after talk	in: A = secondary address	a----	a---	*
TXJUMP	FF6C	---	fe9d	Jump to code at long address AYX		axy-	axy c	
UDTIM	FFEA	---	f979	Part of KB scan (no clockwork!!)	logs keys: enter, +, -, /, stop	----	a---	
UNLSN	FFAE	32e	f2af	Unlisten all IEEE devices		----	a---	*
UNTLK	FFAB	32c	f2ab	Untalk all IEEE devices		----	a---	*
VECTOR	FF84	---	fb9	RAM vectors storing	in: C = 0 moves user list at AYX to vector area C = 1 moves vectors to addr AYX	axy c	axy-	
VRESET	FF6F	---	fbca	Set button-reset code to bank 15 at X, Y	in: X = low Y = high	-xy-	a---	

Unmasking The Kernal

John Russell
St. John's, NF

- A collection of notes about using the I/O routines

Every programmer who takes up machine language soon runs into those puzzling phenomena known as the "Kernal Routines". Those who have already attained enlightenment use them with abandon and urge others to do the same, but they can be pretty darn confusing if you have barely passed the "LDA #\$00" stage. At least, they were to me, armed as I was with only a C64 Programmer's Reference Guide and a copy of Supermon.

I was by no means an overnight success at learning machine language. At first I shunned the machine language section of the Guide as if I was afraid the pages would bite me. But older, wiser heads assured me that if I progressed to the point where I needed speed beyond that of Basic, I would have to become involved with machine language. I was tough to convince, but eventually I was gripped with curiosity.

And so one day I decided to give M-L a try. I studied carefully the explanations and examples from the Reference Guide. I fiddled with Supermon. I printed the alphabet. I changed the colour of the screen, and finally figured out what numbers such as \$D020 stood for. I memorized the mnemonics and their functions. I printed the alphabet again. But it was here that I was stopped cold, because I couldn't really cause anything to happen (aside from changing the screen colours and the ever-popular alphabet printing). I needed two things: y-indexed loops and Kernal Routines. The former were described well enough in the Guide, I just had to think a bit about where to use them; the Kernal Routines, however, were downright confusing. This might not seem like a major crisis - who needs them, anyway? What exactly do they do? Well, the easiest way to explain it is to say that without them, the only way to make the computer communicate with the outside world (i.e. you or another user) is by poking information either to screen memory or to the mysterious "CIA chips" (one bit at a time!). This did not seem to me to be a real possibility, so I set about to figure out exactly what the Kernal was all about.

Don't laugh, but I thought that Supermon would understand the names of each routine, so I had commands like "JSR CHROUT" without benefit of an assembler (Supermon, Micromon et al, with their narrow format for entering instructions, are monitors). I couldn't figure out the order of many sequences. There are a host of routines that can be used to access the disk or printer, and just to be safe, I would always use as many as I could. Needless to say, it was a nice while before I could do anything with the drive or printer.

My biggest beef was that many important routines, like "CLRCHN" are not identified as important, and routines like "LISTEN" (which I have never once needed to call directly) are not identified as unnecessary. How could that be a problem, you ask? I "listened" and "unlistened" my drive to death but never did a "CLRCHN", so I always ended up with a locked up computer, an error light, and a star file in my directory.

The universal input/output routines are not identified as such, so I could never decide which one was the right one (CHROUT and GETIN are always best, with one exception - simulating a Basic INPUT statement). Also, there was no step-by-step guide to opening a file and performing I/O functions. I ran out of stack space in my head trying to follow all those preparatory routines back to the beginning. And putting "s,w" or "p,w"

in the filenames in order to write anything to the disk escaped me for quite some time.

An extreme case? Learning disabilities? I was beginning to suspect that such had to be true. In contrast to my quick grasp of commands and techniques in Basic, I was a snail at learning machine language.

Looking back, I can see that it wasn't really that bad. Once I learned the op-codes and the different types of indexing I could do anything I wanted to, limited only by my knowledge of the 64's input/output chips. Now when I'm asked questions about machine language by learners, I find I'm able to give detailed answers without referring to memory maps or the Guide - because I was forced to figure such things out for myself and test them by trial and error.

I suspect that everyone has similar problems, at least starting off. And accessing the Kernal Routines properly is likely to be the highest hurdle you'll have to clear on your way to becoming a proficient M-L programmer.

So: here's my guide to using Kernal Routines.

- Learn the hex addresses of the important routines. Using the symbolic labels in an assembler such as PAL is all very well and good, but you'll have difficulty understanding disassemblies or other people's code otherwise. I find "jsr \$FFD2" just as easy to type and recognize as "jsr CHROUT".

- However, those who have access to an assembler can save themselves time and bother by assigning labels to important routines early in a program (eg OPEN = \$FFC0). For those who do programming which makes extensive use of the Kernal, save a PAL symbol table or a Library file (for the Commodore Macro Assembler) which consists of nothing but the labels and addresses of frequently used Kernal routines. Use a ".lst" in PAL or ".lib" in the Macro Assembler to have the labels assigned automatically.

- To print a character to any device (including screen) use CHROUT (\$FFD2). No need to ever use CIOUT, LISTEN, or SECOND, because the routine at \$FFD2 checks to see if the character is going to disk or printer and calls these routines when they are necessary.

- To receive a character from any device (including keyboard) use GETIN (\$FFE4). This does away with the need for ACPTR, CHRIN, TALK, UNTLK, and TKSA, for the same reasons as above.

- Of course, this means that any time you want to send to or get from disk, printer, modem, etc. you must first indicate this to the computer (Aha! you say). After a file has been opened, use CHKIN or CHKOUT to select that file for the proper operations. To avoid serial bus confusion, use CLRCHN first. It's easier to remember if you put in a routine called "toprinter" or "fromdisk" which will call CLRCHN, then CHKIN or CHKOUT whenever necessary (or if you're not sure where your data is going to or coming from).

– The friendliest routine to use both before and after doing disk or printer work is CLRCHN (\$FFCC). Anytime you perform input or output to peripherals, use this afterwards to make sure input and output go back to keyboard/screen. Use it often; you can never be too careful.

– Use CHRIN only as the equivalent of a Basic input statement, and in exactly the way it's shown in examples. This is a bizarre routine which can have unpredictable results if not handled carefully. Try calling it a single time if you're not convinced.

– When using LOAD and SAVE, don't use OPEN first. They perform OPEN automatically.

– CLALL doesn't always do what you'd expect. Use the CLOSE routine on each individual file. If you're lazy (like me), have several files open, or have a program crash with the drive light on and file number unknown, OPEN the disk command channel, then CLOSE it. This closes all open files to the disk, so never CLOSE the command channel if you're not finished with your other files.

– If you are sending disk commands, using block read or write, checking the error channel, or anything that requires you to have the command channel to the disk open, keep it open for the duration. It can't hurt, and you will suffer no ill effects if it isn't closed at the end. There will, however, be nasty surprises if you close it before finishing up with your other files.

– Sending a command to the disk is most easily done by setting the filename to be the command (e.g. "s0:test") before performing the OPEN. Then simply open the command channel. If no command is to be set, use a filename of length zero. Disk commands after the first must be printed to the channel, as in the Basic command print#15, "i0".

– The READST routine (\$FFB7) is used mainly to detect the end of a file (this is the cryptic "EOI line" mentioned in the C64 Guide; it stands for "End Or Identity"). A loop using it would look something like this:

```
getin = $ffe4
chrout = $ffd2
readst = $ffb7
loop = *
    jsr getin      ;get a char
    jsr chrout     ;print it
    jsr readst     ;check status
    and #$40       ;is bit 6 (EOI) still clear
    beq loop       ;yes, go back
```

– When reading the error channel, a loop must be used to get characters until a return (chr\$(13)) is received in order to turn off the error light.

– If you get "searching for..." and similar messages when loading and saving with machine language you can turn them off with "asl \$9d". This clears a certain bit and avoids the hassle of the SETMSG routine.

– Use drive and file-type declarations in filenames to avoid errors – don't call it "test", call it "0:test,p,w". The exceptions to this rule are LOAD and SAVE, which don't require file type to be specified. It is, however, possible to get a look at a sequential file by doing a LOAD of "name,s,r". The resulting program is somewhat garbled, but it can be a useful time-saver.

A few examples (in PAL format) should serve to make things more clear. Users of M-L monitors should use actual numbers and addresses in place of labels.

To open program file "test" on disk for reading. . .

```
;**** below is equivalent of: 'open 8,8,8,"name" ' *****
lda #$08      ;the file number the computer refers to
ldx #$08      ;device number 8 – the drive
ldy #$08      ;secondary address 8 – anything other than
               ;15, 0, or 1 is safest
jsr $ffba     ;setlfs – use the above numbers
lda #$0a      ;10 chars in filename
ldx #<name    ;the $34 in an address like $1234
ldy #>name    ;the $12 in an address like $1234
jsr $ffbd     ;set the file's name
jsr $ffc0     ;do the actual opening
ldx #$08      ;file #8 (NOT device 8)
jsr $ffc6     ;chkin – ignore keyboard, receive characters from file #8
```

(routine using \$FFE4 to read from file)

```
;**** close file # 8 ****
jsr $ffcc     ;finished with disk for now
lda #$08      ;file #8 again
jsr $ffc3     ;close file #8
rts           ;and we're done
```

```
;
name = *       ;filename goes below
.asc "0:test,p,r"
```

```
;
;NB: users of monitors must poke their
;   filenames into memory and figure
;   out the hex addresses themselves
```

To save memory from \$1234 to \$5678 as "prog" on disk. . .

```
start = $1234   ;start of ram to save
end = $5679     ;end of ram + 1
;
lda #$08
tax
tay
jsr $ffba      ;3 8's, as in above example
lda #$06       ;6 chars in filename
ldx #<name     ;lower 2 hex digits of address
ldy #>name     ;higher 2 hex digits
jsr $ffbd      ;set the filename
lda #>start
sta $fa        ;start of save
lda #<start    ;goes into $fa,$fb
sta $fb        ;as lo-byte, hi-byte
lda #$fa       ;because $fa was used above
ldy #<end      ;one more than
ldx #>end      ;end of save
jsr $ffd8      ;now save it
rts
;
name = *       ;could be anywhere
.asc "0:prog"   ;no file type needed for load & save
```

Notice that when save was used, there was no need to open a channel or set an output device. Doing so can lead to a file with file type "del" in the directory.

Of course, it's impossible to cover all aspects of the Kernal routines in a single article – you could write a sizable manual on their intricate workings. The best bet is for programmers to refer to a work like the Programmer's Reference Guide (hence the name) for detailed how-to-use information and the quirks each routine has. And, of course, keep reading The Transactor!

Kernal Who?

Evan Williams
Williams Lake, BC

I started computing when FORTRAN IV was popular and the machine filled the basement of one of the halls on campus. Interactive was a word used in teaching, not in programming. All jobs were run in batch mode; you punched your card deck, dropped it off at midnight, and came back at 3 AM to get the printout. A few of the well-equipped physics labs had a PDP-4 model or two, a mini-computer. In those days "mini" meant really small (like skirts). The PDP series was lucky to have 8k of memory installed, and many had 4k. I dropped out of computing for a number of years, not being able to afford 2-3 million for a computer or a van to port it around in. Then came PET. I eagerly laid out \$1200 dollars for one of the first ones to hit B.C. back around 1978 and took it home to see what it could do. I soon discovered the *great* feeling of being able to change a program the moment a mistake was discovered. This was interactive programming. To add to this, the PET has one of the best screen editors ever implemented (long before IBM PC or TI Professional). One thing led to another, and soon I was searching for a way to boost performance. There is only one really practical way to do this, and that is machine language.

What's a KERNAL?

Machine language is SUPER SPEEDY compared to BASIC, but you have to do every little thing yourself and it is difficult to make your routines as flexible as BASIC. Fortunately, there is help, and it resides within the ROMs included with your computer. This help is in the form of a useful set of general-purpose machine language routines that perform a variety of functions. These functions are primarily input, output and internal housekeeping. The routines are set up so that they will access the correct device, passing information back and forth from your program in a simple and predefined way. This block of machine language routines is the heart of the computer's operating system, and in the C-64 has a ROM chip all of its own to live in. Around these routines are built many of the functions utilized by BASIC or most any other program written in machine language. Because the routines are such a central element of the computer, the name KERNAL is applied. Many programming languages exist that have a core of predefined routines that may be used to assemble more complex functions. In these languages the central core routines are often called the KERNAL. The method of using these routines in Commodore computers is the same, regardless of which model you own. To Commodore's credit, the requirements for entering each routine and the results have been kept as constant as possible on different machines.

The Jump Table

The jump table is the way by which all KERNAL routines should be called. This table is a sequence of ML jump instructions found near the top of memory. They are in the same place in all Commodore computers except for routines that are unique to a specific machine. To call a KERNAL routine one must first call any prerequisite routines and then load certain registers with byte values needed to transfer

required information to the KERNAL routine. Some routines need no setup at all, while others need one or two previous routine calls to get things ready. For the purposes of this discussion we will use the C-64 jump table as an example. Those of you using other models of Commodore computers, particularly the PET series, will find the jump table similar, generally containing a subset of the C-64 jump table. If you have a machine language monitor handy for your computer, (such as supermon, micromon, etc.) you should have a look at the section of memory starting at \$FF81. When this area is disassembled, you will find a sequence of "JMP \$xxxx" instructions where "\$xxxx" is an address of either a routine in the ROM or of an indirect jump vector.

In the case of an indirect vector the address of the actual routine will be contained in two consecutive memory locations starting with the one referenced by the "JMP" instruction. The target address is stored in the vector location with the low order byte in the low order memory address and the high byte in the high order memory location. For example, the CHROUT (\$FFD2) routine (see table I) vectors through location \$0326 by means of a "JMP (\$0326)" instruction in the jump table. When this instruction is executed, the microprocessor will fetch the two bytes contained in locations \$0326 and \$0327 and install them in the program counter. Program execution will then continue starting at this new address.

A very important feature of the routines using an indirect jump vector is the fact that the vectors are stored in R/W memory (usually known as RAM even though ROM's are also Random Access Memory). This means that the programmer may change these vectors to point to his own routines, or simply to a RTS instruction so as to disable a routine. "Patching" these vectors is a simple and effective way of adding or modifying the operation of the computer operating system. As an example, new commands may be added to BASIC by changing the error message vector (\$0300) to point to your routine. This routine would then use the CHRGET routine to re-get the offending statement and compare it to your list of valid commands. If no match were found then the accumulator value would be restored and control passed on to BASIC by JMPing to the normal error routine (\$E38B). Another technique for changing KERNAL vectors, particularly those using direct JMP commands, is to copy the KERNAL to the underlying RAM and switching off the KERNAL ROM. It is now possible to change anything you wish in the KERNAL including any and all jump table vectors.

Using The KERNAL

Most of the KERNAL routines require some form of setup before being called. Some, however, do not. Generally, the routines that reset or clear something do not require any preparation. For example, the RESTOR routine does not require any prior KERNAL calls or register setup. When called, it will rewrite the jump table vectors starting at \$0300 to the default values. An idiosyncrasy of the manner in which

this routine is written causes it to also write the jump table vectors to RAM locations \$FD30-\$FD4F. This happens because the routine VECTOR uses part of the same code and is written to allow moving of the vectors. As a result, if you have anything important stored in \$FD30-\$FD4F it will be over-written when RESTOR is called. RESTOR is one of the routines called when the keys RUN/STOP-RESTORE are pressed. (You will notice these bytes being over-written if you view a high-res screen at \$E000, then RUN/STOP-RESTORE and view it again.)

More commonly, the KERNAL routines require some register setup before use. Probably the most-used such routine and perhaps the most complex is CHROUT (\$FFD2). This routine is used to output a single character. The byte value of the character is placed in the accumulator with a LDA XX command and CHROUT is called using JSR \$FFD2. If printing on the screen is desired, no other setup will be required. The CHROUT routine will examine several flags to find out what channel is to be used. If no channels are open or enabled, CHROUT will then examine the byte value and determine if it is a control character.

If not, the correct screen location will be calculated, the character translated to the correct screen code, reverse printing checked, the value placed in screen memory, the color memory updated, screen scrolling checked/done, line link table updated and more. When done, CHROUT and all other KERNAL routines return with an RTS command.

The nice part is that the programmer does not have to worry about any of this. The KERNAL does it for you. Another well used routine is GETIN. This functions in almost the same manner as the BASIC GET statement. When GETIN is called, with no channels open, a single character from the keyboard buffer will be returned in the accumulator. If the keyboard buffer is empty, the value zero will be returned.

PROGRAM 1

Program 1 is a simple input routine in machine language which uses the GETIN routine to fetch characters from the keyboard buffer. The program exits when a return is pressed. Only alpha-numeric input is allowed with no control or cursor characters recognized or printed. Unlike the BASIC input routine this program will accept up to 255 characters as input. It is not possible to move off the line as only the delete key may be used to edit. When the return key is pressed, the program alters the pointer of the first BASIC variable declared by the BASIC program to point at the input buffer. For this reason, the first variable declared in the BASIC program should be a string variable. Upon return from this program this first variable will contain the input data. The length of input may be controlled by the second variable declared in the BASIC program. This must be an integer variable with a value range of 0-255. Program 1 as written will use \$C000 to \$C0FF as the input buffer. See table II for entry address and examples of use.

Program 1 gets all input from the keyboard and puts all output to the screen. This is because no input or output channels other than the default ones have been specified. To get information from some channel or device other than the keyboard, it is necessary to call some preparatory routines first. Similarly, to output to a device other than the screen, some setup is required.

PROGRAMS 2 and 3

Program 2 is a low resolution screen dump routine. This program opens a file to disk and writes the contents of the screen including

sprite pointers on the disk. It functions by first obtaining the location in memory of the first variable declared in a BASIC program and using the contents of this string as the file name/command string to send to the disk drive. The KERNAL routine SETNAM is then called to set up system pointers to this string.

The logical file number, the device number and the secondary address are then loaded in the correct registers and the SETLFS routine is called. At this point all that remains is a call of the OPEN routine and we have a open file. This sequence produces exactly the same result as the BASIC statement OPEN 1,8,2,"0:test,s,w" assuming the string variable passed from BASIC to program 2 contained "0:test,s,w". The file name/command string may be located anywhere in memory. All that is necessary is to place the low/high values of the start address of the command sequence in the .x and .y registers along with the length in the .A register (Accumulator) and call the SETNAM routine.

Next in program 2 the CHROUT routine is directed to the device used by file #1 by loading the .x register with the file number and calling the CHKOUT routine. This means that any calls of CHROUT will send the byte in the accumulator to the disk. All values (0-255) are written with no filtering of exceptions. After reading all bytes from the screen and sending them, the accumulator is loaded with the file number and the CLOSE routine is called. This closes the file and notifies the disk drive that it is the end of the file. The CLALL routine is called next; this resets all I/O to the default channels and clears the file table.

Program 3 is the screen read routine and is very similar to program 2. The main difference is that the file opened is a read file (0:test,s,r) and the channels opened are input channels. It should be noted that a channel may be enabled for input using CHKIN without affecting the output channel when calling CHROUT. The same is true for output. It is therefore possible to input from disk using GETIN and print to the screen using CHROUT. Conversely, input from the keyboard using GETIN and output to disk or printer using CHROUT is possible. An important difference in program 3 is the use of a temporary storage location for the index variable used in the .y register. This is necessary because all registers are clobbered by the GETIN (\$FFE4) routine (see TABLE I).

The file opened in programs 2 and 3 is exactly the same as a file opened by a BASIC program. Because of this, if a machine language program is to be used as a subroutine of a BASIC program, it is perfectly okay to open and close the file within the BASIC portion of the program. It is recommended that when fetching data from the disk the GETIN routine be used instead of the CHRIN routine. The reasons are the same as for using GET in BASIC instead of INPUT in that GETIN will accept any and all characters. The nice thing about machine language is that GETIN will work just as fast as the CHRIN routine when all the overhead of BASIC is absent. Program 3 does not test for a valid character after calling GETIN since the file length is always constant.

PROGRAM 4

Program 4 is the most complex. This program provides instant checking of the disk error channel when the Commodore and control keys are pressed together. The disk status is printed on the top line of the screen and the cursor position is maintained. If it is desired to call this program from within another program, the last instruction can be changed to a RTS, the error check BCS EXIT changed to NOP's, and the program called at BEGIN.

Program 4 uses the serial bus communication routines in the KERNAL without opening a file. The serial device is commanded "speak" with the TALK (\$FFB4) routine and the secondary address sent using the TKSA (\$FF96) routine. The *Commodore Programmers Reference Guide* incorrectly states that to send the secondary address one loads the accumulator with the secondary address value and calls the routine. The GUIDE does not mention that the secondary address value must first be OR'ed with the hex value \$60. To access the disk command channel, secondary address \$6F must be sent (\$0F OR \$60 = \$6F). This applies only to the direct serial secondary address routines TKSA and SECOND. The SETLFS routine requires only the unmodified correct secondary address value.

When program 4 is run, the interrupt vector at \$0314-\$0315 is set to point at the main body of program 4. Sixty times per second the system interrupt causes the code at START to execute. If a match between location \$028D and \$FE is found, the rest of the program will execute. If no match is found the normal interrupt routine is JMP'ed to. When the program executes a new value is placed in the countdown timer \$FE and the TALK (\$FFB4) routine is called with a device number of eight. Following, the TKSA (\$FF96) routine is called with secondary address \$6F. This opens the command channel in the disk drive for talking (to the computer).

Next the PLOT (\$FFF0) routine is called with the carry bit set. This returns the row and column position of the cursor in the .x and .y registers. These are stored. The cursor is now set to the home position by a call to CHROUT with a value of \$13 in the accumulator. Next the status byte (same as ST in BASIC) is set to zero.

The ACPTR (\$FFA5) routine is then called and the returned value printed on the screen by CHROUT. A call to READST (\$FFB7) follows to determine if an EOI was sent. If not, the program loops to NEXT and repeats until READST returns a non-zero value. This indicates the disk has said all it is going to and isn't speaking to us anymore.

We then try to reset the cursor back where it came from and run into a problem in the PLOT routine. Unfortunately, the values returned when PLOT is called to obtain cursor position are not always the same as the values we must use to restore cursor position. The problem comes up when the cursor is positioned on a wrap line, ie. one that is longer than 40 characters. The restore position part of the PLOT routine incorrectly handles the row calculations. We must test the column position returned by PLOT and if it is greater than \$27 (decimal 39) we must subtract \$28 before calling PLOT.

Finally we call the UNTLK (\$FFAB) routine to untalk the serial bus devices and then JMP to the normal interrupt handler.

PROGRAM 5

Program 5 is a BASIC program that generates a machine language program file on disk. This file will contain all four example programs with the call addresses as listed in TABLE II. Just type it in and place a ready to use disk in the drive. Then RUN the program. The ML program can be loaded with LOAD "KERNAL WHO C100",8,1.

Problems and tips

A few problems have been noted when using the KERNAL routines to talk to the disk drive. If a UNTLK is followed immediately by a TALK command the computer may "crash". This appears to be caused by the drive not being able to respond to another command until it has finished some internal work thus causing it to miss the attention

sequence from the computer. This takes a few milliseconds and it is best to wait at least 100 milliseconds before sending a new command. This is one reason for the countdown timer logic in program 4. That logic is also implemented since program 4 is executed on the interrupt and these interrupts occur every 16.7 milliseconds. It is necessary to prevent the routine from being called by a second interrupt that occurs while the routine is still executing. In most interrupt driven routines this is not a problem but the serial bus communication routines have a nasty habit of clearing the interrupt flag thus allowing further interrupts.

Something that is not specifically mentioned in the GUIDE is the fact that the KERNAL routines never call any routines in the BASIC ROM residing at \$A000-\$BFFF. This means that the BASIC rom may be turned off and the KERNAL used as much as you like. This allows you to use the 8k of RAM there for your own programs. Keep in mind that the BASIC ROM makes frequent use of the KERNAL so the converse is not true.

Another system characteristic that is not mentioned anywhere is that sprites MUST be turned off when the serial bus is used. It is not sufficient to hide them on the edge of the screen, they must be OFF! The VIC chip steals time from the 6510 cpu when sprites are turned on and this will clobber the serial bus timing routines, particularly the EOI (end of information) detection. Your computer will occasionally miss the EOI signal and then wait until the sun burns out to get it. Using sprites does not seem to affect the RS-232 routines since the timing windows are much wider.

When using the system RS-232 routines it is only necessary to OPEN a file to device number two and output with CHROUT or input with GETIN after setting the correct I/O channels. Do not use CHRIN with RS-232 since CHRIN is dependent on receiving a carriage return to terminate the routine. Also, the RS-232 interrupt system uses the non-maskable interrupt (NMI) as does the serial bus. Therefore, you cannot send or receive RS-232 data and serial bus data at the same time.

Another item of interest is the way the CLOSE routine handles the RS-232 channel. If CLOSE is called then the RS-232 file is killed along with the buffers. The user port will be set to default I/O values. If the RS-232 file is to be closed without affecting these things call the CLALL routine instead. This will wipe out the file table and set default I/O but the user port is unaffected and the buffers remain allocated. One problem with RS-232 is that when a RS-232 file is OPENed, the user port is set to a standard default condition. This means that if you were using some of the pins for certain non-implemented functions, such as telephone line control, your output conditions and port values may be disrupted. The only way to deal with this is to open the RS-232 file "manually". To do this you will have to setup all the table and file flags, allocate buffers, set interrupt timers etc. yourself. A full description of this process is beyond the scope of this article but may be the subject of a future article.

Full details of the entry, exit and error conditions to be considered are in the *Commodore 64 Programmers Reference Guide*. I have listed the KERNAL routines in TABLE I in order by address because the table in the GUIDE is in alphabetical order by label and that is very inconvenient when you are looking through a disassembly and trying to find out what routine is used.

Using the KERNAL is not difficult. Many of the operations are not much different from the way BASIC works. It saves time and will make life a lot easier for the programmer.

TABLE I

Commodore 64 KERNAL Jump Table In Address Order

Label	Call addr	Vector addr	Target addr	Function Description	Register Usage		
					entry	return	used
IOINIT	\$FF84	-	\$FDA3	initialize i/o	- - -	- x y	a x y
RAMTAS	\$FF87	-	\$FD50	ram test	- - -	- - -	a x y
RESTOR	\$FF8A	-	\$FD15	restore vectors	- - -	- - -	a x y
VECTOR	\$FF8D	-	\$FD1A	move vectors	- x y	- - -	a x y
SETMSG	\$FF90	-	\$FE18	control kernal msg	a - -	- - -	a - -
SECOND	\$FF93*	-	\$EDB9	send listener second	a - -	a - -	a - -
TKSA	\$FF96*	-	\$EDC7	send talker second	a - -	a - -	a - -
MEMTOP	\$FF99	-	\$FE25	set top ram pointer	- x y	- x y	- x y
MEMBOT	\$FF9C	-	\$FE34	set start ram point	- x y	- x y	- x y
SCNKEY	\$FF9F	-	\$EA87	scan keyboard	- - -	- - -	a x y
SETTMO	\$FFA2	-	\$FE21	set IEEE timeout	a - -	- - -	a - -
ACPTR	\$FFA5*	-	\$EE13	input serial byte	- - -	a - -	a x -
CIOUT	\$FFA8*	-	\$EDDD	output serial byte	a - -	a - -	- - -
UNTLK	\$FFAB*	-	\$EDEF	untalk serial bus	- - -	a - -	a - -
UNLSN	\$FFAE*	-	\$EDFE	unlisten serial bus	- - -	a - -	a - -
LISTEN	\$FFB1*	-	\$ED0C	listen serial device	a - -	a - -	a - -
TALK	\$FFB4*	-	\$ED09	serial device talk	a - -	a - -	a - -
READST	\$FFB7	-	\$FE07	read i/o status byte	- - -	a - -	a - -
SETLFS	\$FFBA	-	\$FE00	set-up logical file	a x y	- - -	a x y
SETNAM	\$FFBD	-	\$FDF9	set file name	a x y	- - -	a x y
OPEN	\$FFC0 (\$031A)		\$F34A	open a logical file	- - -	a - -	a x y
CLOSE	\$FFC3 (\$031C)		\$F291	close a single file	a - -	a - -	a x y
CHKIN	\$FFC6 (\$031E)		\$F20E	enable input channel	- x -	a - -	a x -
CHKOUT	\$FFC9 (\$0320)		\$F250	enable output chan	- x -	a - -	a x -
CLRCHN	\$FFCC (\$0322)		\$F333	set chans to default	- - -	- - -	a x -
CHRIN	\$FFCF (\$0324)		\$F157	input characters	- - -	a - -	a x -
CHROUT	\$FFD2 (\$0326)		\$F1CA	output a character	a - -	a - -	a - -
LOAD	\$FFD5*(\$0330)		\$F49E	load to memory	a x y	a x y	a x y
SAVE	\$FFD8*(\$0332)		\$F5DD	save from memory	a x y	a - -	a x y
SETTIM	\$FFDB	-	\$F6E4	set jiffy clock	a x y	- - -	a x y
RDTIM	\$FFDE	-	\$F6DD	read jiffy clock	- - -	a x y	a x y
STOP	\$FFE1 (\$0328)		\$F6ED	test stop key	- - -	a - -	a x -
GETIN	\$FFE4 (\$032A)		\$F13E	get char from chan	- - -	a - -	a x y
CLALL	\$FFE7 (\$032C)		\$F32F	clear/close files	- - -	- - -	a x -
UDTIM	\$FFEA	-	\$F69B	update jiffy clock	- - -	- - -	a x -

Notes:

Registers indicated as being used in the "used" column may contain the same value loaded to call the routine. If no usage is indicated, then the register is safe to use for other purposes, eg. indexing, counting, storing, etc. When GETIN is called for RS-232 input, the .x and .y registers are not affected. The processor status register and the accumulator carry bit are affected by nearly all KERNAL routines.

When sixteen-bit values are passed in or out of a routine, the .x register contains the low byte and the .y register contains the high-order byte.

Detailed information on using specific routines may be found in the Commodore 64 Programmer's Reference Guide.

* Serial I/O routine only, not compatible with some IEEE-488 adapters

** These routines use an indirect jump link AFTER being entered

Program 1

Keyboard input routine using GETIN and CHROUT with 255 character buffer and automatic BASIC variable access. Stores characters at \$C000-\$CFFF. The first variable in the BASIC program should be a string (eg. a\$ = " "). When the program is called this variable will contain the input. The length of input is controlled by the second variable declared by the BASIC program. This variable must be an integer (eg. a% = 10) and have a value of 0-255.

```

LDA #$00      ;a zero in .a
STA $FE       ;index storage
STA $CC       ;flag to flash cursorvariable offset
LDY #$0A      ;variable offset
LDA ($2D),Y   ;get low byte of second var
STA $FD       ;save it
LOOP JSR $FFE4 ;"GETIN", go get a character
CMP #$00      ;is it a zero?
BEQ LOOP      ;if zero then loop
CMP #$0D      ;is it a return key?
BEQ END       ;exit if return pressed
CMP #$14      ;is it a delete key?
BNE NODEL     ;not delete then skip

```

```

LDY $FE       ;buffer pointer
BEQ LOOP      ;loop if buffer empty
DEC $FE       ;delete by decrementing pointer
JMP OUTPUT    ;jmp output
NODEL TAX      ;save .a in the .x register
AND #$7F      ;remove high bit from char
CMP #$20      ;is it a control character?
BCC LOOP      ;if less than #$20 yes so loop
TXA           ;restore character to .a reg
LDY $FE       ;retrieve buffer index
CPY $FD       ;check if limit reached
BCS LOOP      ;carry set, limit reached
INC $FE       ;increment buffer pointer
BNE PUT       ;if buffer not full skip to "PUT"
DEC $FE       ;oops, buffer full so back down one
JMP LOOP      ;loop
PUT STA $C000,Y ;place byte in $c000 buffer
OUTPUT JSR $FFD2 ;"CHROUT" - print the character
LDA #$00      ;zero
STA $D4       ;disable quote mode
JMP LOOP      ;play it again sam
END LDY #$02   ;offset to string variable length
LDA $FE       ;buffer pointer

```



```

STA ($2D),Y ;put in first variable length byte
INY         ;inc index y offset to pointer low byte
LDA #$00    ;low order byte buffer location
STA ($2D),Y ;store it in string variable pointer
INY         ;increment index
LDA #$C0    ;high order byte
STA ($2D),Y ;store it in string variable pointer
INC $CC     ;turn off cursor
LDA #$20    ;a space to clear the cursor block
JSR $FFD2   ;print it,
RTS         ;and return

```

PROGRAM 2

Dump screen contents to disk using name found in first variable declared in BASIC program calling this routine. The first variable should be a string variable or the name may be rather strange.

```

START  LDY #$02 ;offset to string length
        LDA ($2D),Y ;get string length
        BEQ EXIT ;zero length, quit while ahead
        PHA ;save length on stack
        INY ;increment index
        LDA ($2D),Y ;get low address of string
        TAX ;put in .x
        INY ;increment index
        LDA ($2D),Y ;get high order pointer
        TAY ;put in .y
        PLA ;pull length from stack
        JSR $FFBD ;SETNAM: set file name
        LDA #$01 ;logical file #1
        LDX #$08 ;device #8
        LDY #$02 ;secondary address
        JSR $FFBA ;SETLFS: set logical file
        JSR $FFC0 ;OPEN
        BCS EXIT ;if carry set then error out
        LDX #$01 ;file number
        JSR $FFC9 ;CHKOUT: set output channel to file 1
        LDY #$00 ;zero index
        STY $FD ;set pointer low byte
        LDA #$04 ;start of screen high byte
        STA $FE ;set pointer high byte
LOOP    LDA ($FD),Y ;get screen character
        JSR $FFD2 ;CHROUT: output a byte to disk
        INY ;increment index
        BNE LOOP ;not 256 yet?
        INC $FE ;increment high byte of pointer
        LDA $FE ;get pointer high byte
        CMP #$08 ;done four pages yet?
        BCC LOOP ;if carry clear then no
        LDA #$01 ;file number 1
        JSR $FFC3 ;CLOSE: close the file
EXIT    JSR $FFE7 ;CLALL: restore default i/o channels
        RTS ;return to BASIC

```

PROGRAM 3

Load screen contents from disk using name found in first variable declared in BASIC program calling this routine. The first variable should be a string variable.

```

START  LDY #$02 ;offset to string length
        LDA ($2D),Y ;get string length
        BEQ EXIT ;zero length, quit while ahead
        PHA ;save length on stack
        INY ;increment index
        LDA ($2D),Y ;get low address of string
        TAX ;put in .x
        INY ;increment index
        LDA ($2D),Y ;get high order pointer
        TAY ;put in .y
        PLA ;pull length from stack
        JSR $FFBD ;SETNAM: set command string
        LDA #$01 ;logical file #1
        LDX #$08 ;device #8
        LDY #$02 ;secondary address
        JSR $FFBA ;SETLFS: set logical file
        JSR $FFC0 ;OPEN
        BCS EXIT ;if carry set then error out
        LDX #$01 ;file number
        JSR $FFC6 ;CHKIN: set input channel to file 1
        LDY #$00 ;zero index
        STY $FC ;zero index temp
        STY $FD ;set pointer low byte
        LDA #$04 ;start of screen high byte
        STA $FE ;set pointer high byte
LOOP    JSR $FFE4 ;GETIN: get a byte from disk
        LDY $FC ;get index temp
        STA ($FD),Y ;store byte on screen
        INC $FC ;increment index
        BNE LOOP ;not 256 yet?
        INC $FE ;increment high byte of pointer
        LDA $FE ;get pointer high byte
        CMP #$08 ;done four pages yet?
        BCC LOOP ;if carry clear then no
        LDA #$01 ;file number 1
        JSR $FFC3 ;CLOSE: close the file
EXIT    JSR $FFE7 ;CLALL: restore default i/o
        RTS ;return to BASIC

```

PROGRAM 4

Fetches disk status and displays on top line of screen when the control and Commodore keys are pressed together.

```

INIT    SEI ;interrupts off
        LDA #H,START ;high order byte of start
        STA $0315 ;change high order vector
        LDA #L,START ;low byte of start
        STA $0314 ;low byte of vector
        CLI ;interrupts on
        LDA #$06 ;match value
        STA $FE ;save it
        RTS ;return
START  LDA $028D ;load keyboard shift pattern
        CMP $FE ;6 = control + Commodore key
        BEQ BEGIN ;if pressed then do it
        LDA #$06 ;countdown limit
        CMP $FE ;reached yet?
        BEQ EXIT ;if yes then continue
        DEC $FE ;countdown one more jiffy
EXIT    JMP $EA31 ;finish interrupt
BEGIN  LDA #$2D ;45 jiffys (.75 second)
        STA $FE ;countdown location

```



```

LDA #08      ;device number
JSR $FFB4    ;TALK: command disk to talk
LDA #6F      ;secondary address 15
JSR $FF96    ;TKSA: send second
BCS EXIT     ;error abort
SEC          ;set carry bit
JSR $FFF0    ;PLOT: fetch cursor location
STX $FB      ;save it
STY $FC      ;save it too
LDA #13      ;home cursor character
JSR $FFD2    ;CHROUT: print it
LDA #00      ;a zero
STA $90      ;clear the status word
NEXT JSR $FFA5 ;ACPTR: get error channel character
JSR $FFD2    ;CHROUT: print it
JSR $FFB7    ;READST: read status byte
CMP #00      ;if zero
BEQ NEXT     ;get another character
LDX $FB      ;cursor x position
LDA $FC      ;cursor y
CMP #28      ;short line?
BCC GOPLOT   ;if yes go plot
SBC #28      ;subtract 40
GOPLOT TAY    ;move to y register
CLC          ;clear carry
JSR $FFF0    ;PLOT: set cursor back
JSR $FFAB    ;UNTLK: untalk serial devices
JMP EXIT     ;finish

```

TABLE II

Call Address For Programs 1, 2, 3 and 4

PROGRAM	HEX	DECIMAL
PROGRAM 1	\$C100	49408
PROGRAM 2	\$C15F	49053
PROGRAM 3	\$C1A6	49574
PROGRAM 4	\$C1F2	49650

All four programs are stored as a block occupying the space from \$c100 to \$C257. \$C000 to \$C0FF is used as buffer space by program 1.

Sample BASIC Programs

To call program 1

```

10 clr: a$ = " ": a% = 10
20 sys 49408
30 print:print a$

```

To call program 2

```

10 clr: a$ = "0:screen dump,s,w"
20 sys 49053

```

To call program 3

```

10 clr: a$ = "0:screen dump,s,r"
20 sys 49574

```

To call program 4

```

sys 49650

```

Program 5 is a BASIC program that will generate a machine language program on disk. This program contains the above four programs and will have the name "kernal who c100". Load this program with the command:

```
LOAD "kernal who c100",8,1
```

Then type NEW and press return.

PROGRAM 1-4 Generator

```

BO 100 rem object file creator for
KG 110 rem programs 1, 2, 3 and 4
PI 120 rem for article "KERNAL WHO?"
EA 130 rem "Evan Williams 1986
CC 140 print "S place disk in drive and press return."
BL 150 get a$: if a$<>chr$(13)then150
MH 160 print "q ok, please wait"
EP 170 for i = 1to344: read a: ck = ck + a: next
CI 180 if ck<>50143 then print "qq error in data
      statements": end
IL 190 print "q data ok, creating disk program file"
DC 200 open1,8,2,"0:kernal who c100,p,w"
FG 210 restore: print#1,chr$(0);chr$(193);
EK 220 for i = 1to344: read a
CB 230 print#1,chr$(a);
DF 240 next: close1
MM 250 print "q done": end
IH 260 :
EE 270 data 169, 0, 133, 254, 133, 204, 160, 10
AF 280 data 177, 45, 133, 253, 32, 228, 255, 201
PO 290 data 0, 240, 249, 201, 13, 240, 49, 201
DL 300 data 20, 208, 9, 164, 254, 240, 237, 198
EL 310 data 254, 76, 62, 193, 170, 41, 127, 201
NK 320 data 32, 144, 225, 138, 164, 254, 196, 253
PH 330 data 176, 218, 230, 254, 208, 5, 198, 254
IE 340 data 76, 12, 193, 153, 0, 192, 32, 210
AH 350 data 255, 169, 0, 133, 212, 76, 12, 193
NM 360 data 160, 2, 165, 254, 145, 45, 200, 169
OF 370 data 0, 145, 45, 200, 169, 192, 145, 45
DI 380 data 230, 204, 169, 32, 76, 210, 255, 160
FI 390 data 2, 177, 45, 240, 61, 72, 200, 177
ED 400 data 45, 170, 200, 177, 45, 168, 104, 32
GN 410 data 189, 255, 169, 1, 162, 8, 160, 2
EF 420 data 32, 186, 255, 32, 192, 255, 176, 34
NF 430 data 162, 1, 32, 201, 255, 160, 0, 132
DD 440 data 253, 169, 4, 133, 254, 177, 253, 32
LN 450 data 210, 255, 200, 208, 248, 230, 254, 165
AK 460 data 254, 201, 8, 144, 240, 169, 1, 32
HC 470 data 195, 255, 32, 231, 255, 96, 160, 2
HB 480 data 177, 45, 240, 66, 72, 200, 177, 45
FF 490 data 170, 200, 177, 45, 168, 104, 32, 189
DH 500 data 255, 169, 1, 162, 8, 160, 2, 32
MI 510 data 186, 255, 32, 192, 255, 176, 39, 162
MM 520 data 1, 32, 198, 255, 160, 0, 132, 252
LI 530 data 132, 253, 169, 4, 133, 254, 32, 228
FF 540 data 255, 164, 252, 145, 253, 230, 252, 208
JF 550 data 245, 230, 254, 165, 254, 201, 8, 144
AE 560 data 237, 169, 1, 32, 195, 255, 32, 231
PK 570 data 255, 96, 120, 169, 194, 141, 21, 3
PG 580 data 169, 3, 141, 20, 3, 88, 169, 6
MA 590 data 133, 254, 96, 173, 141, 2, 197, 254
DF 600 data 240, 11, 169, 6, 197, 254, 240, 2
MC 610 data 198, 254, 76, 49, 234, 169, 45, 133
JO 620 data 254, 169, 8, 32, 180, 255, 169, 111
MD 630 data 32, 150, 255, 176, 237, 56, 32, 240
DN 640 data 255, 134, 251, 132, 252, 169, 19, 32
JO 650 data 210, 255, 169, 0, 133, 144, 32, 165
KK 660 data 255, 32, 210, 255, 32, 183, 255, 201
CG 670 data 0, 240, 243, 166, 251, 165, 252, 201
FP 680 data 40, 144, 3, 233, 40, 24, 168, 32
OG 690 data 240, 255, 32, 171, 255, 76, 18, 194

```


Adding Functions to Basic

Frank E. DiGioia
Athens, Georgia

Execute Machine Language programs inside your 1541

How would you like to be able to add functions to BASIC with as much ease as you are able to add commands through the use of wedge programs? It can be done. And, in fact, it is just as easy to implement a function wedge as it is to implement any other type of wedge program. The natural question is, of course, if it is so easy, why haven't we seen function wedges before? I think that the reason is simply because the types of wedges we are most familiar with are the ones which are least suitable for adding functions.

If you try to think of a CHRGET wedge or an IERROR wedge returning a function value, then, it does, indeed, look like a tough job, because these types of wedges are not really suitable for returning values. We need a whole new type of wedge. In this article and the two that will follow it, we will explore several different types of wedges which are not in common use, but which have great potential for opening up new avenues of programming for those who desire to enhance the working environment of their computer.

Why A Function Wedge?

Perhaps you're wondering why we even need a function wedge. After all, Commodore was good enough to include the USR function in BASIC 2.0 which allows us to add our own functions to BASIC. While it is very true that we can add almost any function we desire via the USR function, the advantages of a function wedge include the fact that many new functions can be defined at one time as well as the fact that with a function wedge, we are free to determine the number of parameters, method of input, etc. For example, take the program line:

```
10 print !cosh(.5),!sinh(.9),!sec(.12)
```

While the purist may argue that the line could be implemented with USR functions, it would take several lines to implement and would not be nearly as clear as the above line. Further, what if you need multiple arguments like:

```
20 z = !mod(x,y)
```

You could say Z = USR(X),Y I suppose, but it just isn't the same.

Create A New Environment

While the above are good enough reasons for using a function wedge, an even more novel way to use added functions is to change the BASIC programming environment. That is, give the illusion of adding new commands, and changing the capabilities of BASIC. For example, a function named @ that takes two parameters and returns a null string can be used to plot the cursor, thus creating a PRINT@(x,y) statement. And who says we need to enclose the function argument in

parentheses? Suppose we make a function named '\$' which converts from ASCII Hex characters to internal floating point. Then we can execute statements like:

```
10 poke $c000,$b4
20 x = $a000
30 print $d000,peek($d000)
```

or add a function named % for binary and you can

```
20 poke $033c,peek($033c)and%1111
```

Admittedly, the above could be done with a USR function but by using a function named '\$' or '%' we create the illusion of a new operating system.

How To Implement The Wedge

Now that you are fully convinced that a function wedge is a worthwhile endeavor, let us examine how to implement one. It isn't hard at all. The vector we will be changing to point to our evaluation routine is named IEVAL and is located at \$030A/\$030B. In addition to just executing our routine, however, we must tell BASIC whether the result was string or numeric and where to find it. As far as the type goes, storing a zero at location \$0D indicates numeric and tells BASIC to look for the result in the FAC (The floating point accumulator — if you don't know where or what that is, don't worry, there are ROM routines that take care of all of that for you.). Storing a \$FF at location \$0D indicates that the result is of type string and BASIC looks on the temporary string stack to find it. If you don't understand how to set up a string, don't worry, we will use a ROM routine to do this for us, too. Our evaluation routine will default to type numeric and the ROM routines we will use to set up a string will set the type to string so we never have to worry about setting the type flag ourselves. In fact, the only thing out of the ordinary that we will have to do, is to set up the string descriptor if the result of our function is a string. You will see how this is done in the example program.

An Example Function Wedge

At the end of this article is a very useful example of a function wedge that you can use and add your own functions to. It is activated by SYS49152 and is immune to RUN/STOP-RESTORE. Here are the functions we will include and an example of how they are used:

The '@' function: Plot the cursor and return a null string. (Note: more than one @ is allowed in one print statement). Type: STRING

Ex. print@(0,14) " my report ";@(2,12) " by john smith "

The '\$' and '%' functions: Convert HEX and BINARY characters to floating point. Type: NUMERIC

Ex. poke\$d016, peek(\$d016) and %11101111

The '#' function: Convert from LO/HI format to 16 bit floating point. (This routine is included just as an example to show how to convert your results to floating point if they aren't already). Type: NUMERIC

Ex. print #(peek(43),peek(44))

could be used to read the start address of BASIC. To read any address, just PRINT # (LO,HI).

Finally, since all the above functions so far have been kind of off the wall, I will include a parser and some examples of functions similar to what you might add yourself; including a straightforward string function, !DSTAT, and a somewhat serious numeric function, !MOD, which shows that even YOU can do floating point operations from machine language.

The !DSTAT function returns the disk status as a string. The status is cleared once it is read so save it in a variable if you need it.

Ex. (Print A Sequential File)

```
10 open2,8,2,"filename": a$=!dstat: if val(a$)<>0 then
   print a$: end
20 for i=0 to 1: get#2,b$: printb$: i=st: next: close2
```

The !MOD(X,Y) function returns the integer remainder from dividing X by Y. MOD has two sister functions, DIV and FRAC, which return the integer quotient and fractional remainder, respectively.

Ex. hi=!div(x,256)
 lo=!mod(x,256)

 zz=!frac(x,y)

Any functions that YOU may want to add can be included via the ! symbol which immediately causes parsing for the function name to execute.

Although the source code for this example is fully commented, I will briefly discuss some important points for those interested in adding their own functions with this code.

How It Works

The wedge itself is very simple. If not for the ! commands, the wedge would only be a few lines long. When a function name starting with ! is found, control is passed to the parser which looks through the command table for a match and jumps to the corresponding address via RTS. That is, it places the address of the routine (minus one) on the stack and then executes an RTS (at the end of CHRGET) to jump to the routine. The parser code is well worth studying. If you want to add your own functions to BASIC, simply put the name of your function in the TABLE being sure to add \$80 to the last byte of the name. Then put the address-1 in the ADRTAB and you are in business.

The !DSTAT function is straightforward and may be used as a model for adding your own string functions. It talks to the drive and then puts the length of the resulting string in .A and in line 2420 asks BASIC where to put the string. DLOOP2 copies the string there and we end

our function in line 2500 by telling BASIC where the string can be found. (Note that XPLOT (The @ function) returns a null string by reserving space for a string of length zero before setting up the string descriptor.)

Writing numeric functions is easy. Just be sure to leave the final result in the FAC. If you were doing an integer calculation, you can convert the result to floating point in exactly the same way as the # function does it. See lines 2980-3020.

Final Notes

The function wedge is the best way possible to add functions to BASIC. It is immune to RUN/STOP-RESTORE, it is compatible with almost every other utility that I know of including the DOS wedge and Epyx Fastload Cartridge (no, the \$ commands do not conflict) and it provides a natural way to pass the results back to BASIC. Any of these new functions can be used in any way that an old one can be used. I.E. the following statement is perfectly legal:

```
a=$ff0d*%11011+sin(!mod($fabcd,%11101))
```

There are a few very minor limitations to the functions presented here. If you are like me, the first test you will try with the \$ function is PRINT \$ABCDEF. This will result in a syntax error because the BASIC keyword DEF is embedded in the number. Adding a space before the F will solve the problem (The \$ routine ignores spaces). The only other limitation is that the MOD/DIV/FAC group is intended to be used with positive operands only. Don't forget when using the @ function in direct mode that the screen will scroll if you print too close to the bottom. This problem can be easily fixed with a WAIT statement. Try this line:

```
print@(0,14)" my report "@(2,13)" by john
doe "@(24,14)" page 1 " ;:wait198,1
```

I hope that you will be able to use the functions presented here and that this example will provide you with the skills necessary to add any additional functions that you may need. The next article, "Command Wedge", will focus on a wedge for modifying existing BASIC commands. Before going there, try these relatively simple projects:

- (1) Add two functions !HI(X) and !LO(X) which return the hi and lo bytes, respectively, of the number X.
- (2) add a function, !SIZE, which returns the length of the BASIC program currently in memory.
- (3) If you have two drives, make !DSTAT require a device number like !DSTAT8 or !DSTAT9.

HINTS: (Don't read unless you are stuck)

- (1) Try something like this: Look at the definition of LO and HI in terms of !MOD and !DIV above. Use JSR \$AEF1 to get parm into FAC. (This routine takes care of parentheses, etc.) Store FAC at TEMP. (See lines 4190-4210). Get 256 into FAC (See lines 2980-3020) Store at MODLUS (See lines 4260-4280) Set flag for MOD or DIV as required and JMP to line 4300.
- (2) Easy. Subtract address of start of BASIC (found at locations 43/44) from address of start of variables (found at 45/46). Convert this result to FAC.
- (3) Simply replace line 2220 with JSR GETBYT. Then add some checks so that no one can hang it up by giving it a crazy device number.

Have fun!


```

NO 100 rem basic loader for function wedge
FK 110 rem by frank e. digioia 11/14/85
NL 120 rem sys 49152 to activate
GP 130 :
JC 140 for adr = 49152 to 49634:read ml
HG 150 cs = cs + ml:poke adr,ml:next
DJ 160 ifcs<>59800thenprint "error in data"
OB 170 :
FO 180 data 169, 11, 141, 10, 3, 169, 192, 141
IN 190 data 11, 3, 96, 169, 0, 133, 13, 32
AH 200 data 115, 0, 201, 36, 240, 22, 201, 37
HL 210 data 240, 21, 201, 64, 240, 20, 201, 35
HO 220 data 240, 19, 201, 33, 240, 18, 32, 121
ND 230 data 0, 76, 141, 174, 76, 29, 193, 76
OC 240 data 32, 193, 76, 213, 192, 76, 13, 193
EN 250 data 169, 0, 141, 132, 192, 170, 168, 200
ND 260 data 189, 136, 192, 240, 64, 232, 209, 122
DP 270 data 208, 2, 240, 243, 202, 189, 136, 192
NB 280 data 16, 8, 41, 127, 209, 122, 240, 19
NI 290 data 208, 8, 232, 189, 136, 192, 240, 37
PA 300 data 16, 248, 232, 238, 132, 192, 160, 0
II 310 data 76, 63, 192, 200, 152, 24, 101, 122
FB 320 data 133, 122, 144, 2, 230, 123, 173, 132
HN 330 data 192, 10, 170, 189, 153, 192, 72, 189
DD 340 data 152, 192, 72, 96, 0, 76, 8, 175
HB 350 data 77, 79, 196, 70, 82, 65, 195, 68
ND 360 data 73, 214, 68, 83, 84, 65, 212, 0
MO 370 data 142, 193, 145, 193, 139, 193, 159, 192
GP 380 data 162, 8, 134, 186, 138, 32, 180, 255
LJ 390 data 169, 111, 133, 185, 32, 150, 255, 162
GI 400 data 0, 32, 165, 255, 157, 60, 3, 232
LK 410 data 201, 13, 208, 245, 32, 171, 255, 202
HL 420 data 138, 141, 12, 193, 32, 125, 180, 172
LP 430 data 12, 193, 185, 60, 3, 145, 98, 136
IJ 440 data 16, 248, 76, 202, 180, 32, 115, 0
AO 450 data 32, 242, 192, 224, 25, 144, 3, 76
JC 460 data 72, 178, 192, 40, 176, 249, 24, 32
OO 470 data 240, 255, 169, 0, 32, 125, 180, 76
JB 480 data 202, 180, 32, 250, 174, 32, 158, 183
JG 490 data 142, 12, 193, 32, 253, 174, 32, 158
IC 500 data 183, 138, 72, 32, 247, 174, 104, 168
JL 510 data 174, 12, 193, 96, 0, 32, 115, 0
ML 520 data 32, 242, 192, 134, 99, 132, 98, 162
LF 530 data 144, 56, 76, 73, 188, 169, 0, 44
CE 540 data 169, 1, 141, 139, 193, 32, 62, 193
PD 550 data 32, 115, 0, 240, 14, 32, 72, 193
PG 560 data 32, 114, 193, 32, 126, 189, 76, 40
FC 570 data 193, 104, 104, 76, 121, 0, 169, 0
FF 580 data 162, 5, 149, 97, 202, 16, 251, 96
CM 590 data 144, 16, 174, 139, 193, 208, 24, 201
EJ 600 data 65, 144, 230, 201, 71, 176, 16, 56
AG 610 data 233, 7, 174, 139, 193, 240, 4, 201
BI 620 data 50, 176, 12, 56, 233, 48, 96, 201
FB 630 data 65, 144, 206, 201, 91, 176, 202, 76
DA 640 data 72, 178, 166, 97, 240, 15, 72, 174
BA 650 data 139, 193, 189, 137, 193, 24, 101, 97
JL 660 data 176, 4, 133, 97, 104, 96, 76, 126
JL 670 data 185, 4, 1, 0, 169, 0, 44, 169
MJ 680 data 1, 44, 169, 255, 141, 139, 193, 32
NN 690 data 250, 174, 32, 138, 173, 162, 230, 160
BG 700 data 193, 32, 212, 187, 32, 253, 174, 32
DA 710 data 138, 173, 32, 247, 174, 162, 225, 160
IC 720 data 193, 32, 212, 187, 169, 230, 160, 193
BN 730 data 32, 15, 187, 32, 12, 188, 32, 204
AH 740 data 188, 173, 139, 193, 240, 23, 165, 97
MA 750 data 32, 83, 184, 173, 139, 193, 48, 13
JJ 760 data 169, 225, 160, 193, 32, 40, 186, 32
LE 770 data 73, 184, 32, 204, 188, 32, 27, 188
FG 780 data 96, 0, 0

```

Function Wedges Source Code

```

OF 1000 :
BH 1010 :function wedge
JK 1020 :by frank e. digioia
OM 1030 :11/12/85
GI 1040 :
HM 1050 * = $c000 ;convenient start
KJ 1060 :
OJ 1070 chrget = $0073 ;get byte of text
JG 1080 chrget = $0079 ;get same byte
IA 1090 ieval = $030a ;evaluation vector
JN 1100 type = $0d ;type flag
MM 1110 :
GP 1120 init = * ;initialize routine
PD 1130 lda #<fwedge
MD 1140 sta ieval
PE 1150 lda #>fwedge
JC 1160 sta ieval + 1
OH 1170 rts
CB 1180 :
NM 1190 fwedge = * ;this is the wedge
ND 1200 lda #$00 ;flag for numeric
BD 1210 sta type ;set type flag
KD 1220 :
DD 1230 jsr chrget ;see what we've got
DC 1240 cmp #'$' ;hex conversionprint
EJ 1250 beq jump
DD 1260 cmp #'%' ;binary conversionprint
IL 1270 beq jump + 3
PC 1280 cmp #'@' ;plot functionprint
CN 1290 beq jump + 6
GE 1300 cmp #'#' ;the # commandprint
MO 1310 beq jump + 9
AK 1320 cmp #'!' ;use the parserprint
GM 1330 beq parser
BN 1340 :not one of ours
ID 1350 jsr chrget ;set flags again
MM 1360 jmp $ae8d ;use original routine
AN 1370 :
ID 1380 jump = * ;jump table for frns
DA 1390 jmp hex
BP 1400 jmp bin
JK 1410 jmp xplot
CG 1420 jmp expand
MA 1430 :
HP 1440 parser = * ;parse & execute
MI 1450 lda #$00 ;clear all regs
BH 1460 sta count ;and counter
JI 1470 tax
HJ 1480 tay
IE 1490 :
AG 1500 ploop iny ;incr text index
NJ 1510 lda table,x ;get table byte
HH 1520 beq error ;end of table
FJ 1530 inx ;incr table pointer
MA 1540 cmp ($7a),y ;compare with text
OG 1550 bne next ;find next word
IH 1560 beq ploop ;match/keep looking
IJ 1570 :
EN 1580 next dex ;bump .x down once
PN 1590 lda table,x ;end of table wordprint
EI 1600 bpl find ;no/find end of word
CC 1610 and #$7f ;yes/mask flag
ID 1620 cmp ($7a),y ;is it a matchprint
CA 1630 beq found ;hooray!!!
JH 1640 bne x1 ;go back for more
IO 1650 :
BO 1660 find inx ;find end of word
HL 1670 lda table,x ;look for negative
HB 1680 beq error ;end of table
IL 1690 bpl find ;keep looking
KB 1700 :
ED 1710 x1 inx ;point to next word
FJ 1720 inc count ;word # in table
FC 1730 ldy #$00 ;reset text index
BK 1740 jmp ploop ;search some more
ME 1750 :
BJ 1760 found = * ;execution routine
GB 1770 iny ;point to next byte
JJ 1780 tya ;update text pointer
EH 1790 clc
HL 1800 adc $7a
NA 1810 sta $7a
BB 1820 bcc * + 4
GA 1830 inc $7b
GK 1840 :
PF 1850 lda count ;get offset in table
HA 1860 asl a ;multiply by two
LN 1870 tax ;use as index
GA 1880 lda adrtab + 1,x;i byte adr
EP 1890 pha ;as return adr hi

```


AB 1900	lda	adrtab,x	;lo byte adr	FF 2820	txa		;put in .a	EH 3740	lda	incr,x	;get incr in .a
OB 1910	pha		;as return adr lo	EA 2830	pha		;keep it safe	MB 3750	clc		
ND 1920	rts		;execute routine	EP 2840	jsr	chrhrt	;check closing paren	AG 3760	adc	exp	;add exp to incr.
AA 1930 ;				CF 2850	pla		;retrieve 2nd parm	CL 3770	bcs	err1	;overflow error
DN 1940 count	.byte	\$00		IP 2860	tay		;put in .y	MO 3780	sta	exp	;update exponent
MM 1950 error	jmp	\$af08	;syntax error	PC 2870	ldx	len	;retrieve 1st parm	OJ 3790	pla		;retrieve byte to .a
OB 1960 ;				MC 2880	rts			OM 3800 exit		rts	
JJ 1970 ;data tables -- add your own				CF 2890 len	.byte	\$00		IF 3810 ;			
CD 1980 ;routine names and addresses				KM 2900 ;				CP 3820 err1	jmp	oflow	
AM 1990 ;here, be sure to add \$80 to				EN 2910 ;				JE 3830 incr	.byte	\$04,\$01	
CF 2000 ;last character of name and				NG 2920 ;the #(lo,hi) command -- convert				LH 3840 flag	.byte	\$00	
JD 2010 ;subtract 1 from the address				EN 2930 ;lo/hi to 16 bit number.				AI 3850 ;			
KF 2020 ;				CP 2940 ;				KI 3860 ;			
CB 2030 table	.byte	'mo','c4','ra','c3		BI 2950 expand	=	*		CO 3870 ;div/mod/frac -- these routines respectively			
JC 2040	.byte	'di','d6','dsta','d4,\$00		KD 2960	jsr	chrget	;get next byte of text	BI 3880 ;return the integer-quotient,			
IH 2050 ;				OG 2970	jsr	getprm	;get parms into x/y	IL 3890 ;integer-remainder, or fractional			
LK 2060 adrtab	.word	mod-1,frac-1,div-1,dstat-1		JO 2980	stx	\$63	;lo byte in \$63	LE 3900 ;part of the quotient a/b.			
MI 2070 ;				BO 2990	sty	\$62	;hi byte in \$62	ML 3910 ;			
GJ 2080 ;				LD 3000	ldx	#\$90	;set exponent to 15	GL 3920 exp	=	\$61	;adr of exp of fac
EG 2090 ;function calculation routines				PE 3010	sec		;don't invert mantissa	LN 3930 facarg	=	\$bc0c	;copy fac to arg
KK 2100 ;				JK 3020	jmp	\$bc49	;convert to fac	EN 3940 facmem	=	\$bbd4	;store fac at adr in (x/y)
LC 2110 ;dstat function				ME 3030 ;				LA 3950 mdiv	=	\$bb0f	;divide fac by mem
OL 2120 ;				GF 3040 ;				CI 3960 subtrt	=	\$b853	;subtract fac from arg
MP 2130 acptr	=	\$ffa5	;get byte from serial port	OO 3050 ;hex/binary conversion routine --				MI 3970 mmult	=	\$ba28	;mult fac by mem (a/y)
BL 2140 fa	=	\$ba	;device number	BP 3060 ;this routine converts ascii				CM 3980 facint	=	\$bccc	;convert fac to integer
BK 2150 sa	=	\$b9	;secondary address	AE 3070 ;hex or binary numbers to floating				MA 3990 round	=	\$bc1b	;round the fac
MD 2160 wbuf	=	\$033c	;buffer for string	LL 3080 ;point.				FC 4000 add5	=	\$b849	;add .5 to fac
DE 2170 talk	=	\$ffb4	;tell device to talk	II 3090 ;				MM 4010 frmnum	=	\$ad8a	;get numeric parm into fac
KC 2180 tksa	=	\$f96	;send 2nd adr for talk	EJ 3100 addbyt	=	\$bd7e	;add .a to fac	KC 4020 ;			
LK 2190 untalk	=	\$ffab	;free serial bus	GA 3110 illegal	=	\$b248	;illegal quantity	ED 4030 ;			
OA 2200 ;				MJ 3120 oflow	=	\$b97e	;overflow error	ON 4040 div	=	*	;entry for div
CC 2210 dstat	=	*		AO 3130 exp	=	\$61	;exponent of fac	DD 4050	lda	#\$00	;flag for div
KM 2220	ldx	#\$08	;device number (disk)	KL 3140 ;				NB 4060	.byte	\$2c	;skip next instr
AE 2230	stx	fa	;first address	NC 3150 hex	lda	#\$00	;flag for hex	EL 4070 mod	=	*	;entry for mod
EH 2240	txa			HM 3160	.byte	\$2c	;skip next instr.	GH 4080	lda	#\$01	;flag for mod
GD 2250	jsr	talk	;tell drive to speak	EO 3170 bin	lda	#\$01	;flag for binary	LD 4090	.byte	\$2c	;skip next instr
OH 2260	lda	#\$6f	;channel 15 (or \$60)	EA 3180	sta	flag	;save flag	CI 4100 frac	=	*	;entry for frac
HK 2270	sta	sa	;secondary address	IJ 3190	jsr	zero	;set fac to zero	EK 4110	lda	#\$ff	;flag for frac
NJ 2280	jsr	tksa	;send it to drive	GJ 3200 ;				JO 4120	sta	flag	;set the flag
BH 2290	ldx	#\$00		EE 3210 loop	jsr	chrget	;get next char.	IJ 4130 ;			
CH 2300 ;				FE 3220	beq	cdone	;end of statement	ND 4140 ;get first parm in fac and 2nd			
KH 2310 dloop	=	*	;read command channel	NF 3230	jsr	convrt	;convert from ascii	BB 4150 ;parm in arg.			
IG 2320	jsr	acptr	;get byte from drive	AO 3240	jsr	incexp	;incr. fac exponent	GL 4160 ;			
EH 2330	sta	wbuf,x	;store character	MI 3250	jsr	addbyt	;add the byte to fac	KH 4170	jsr	chkflt	;open parenprint
AA 2340	inx			OI 3260	jmp	loop		BG 4180	jsr	frmnum	;get first value
BK 2350	cmp	#\$0d	;carriage returnprint	MD 3270 ;				PD 4190	ldx	#\$<temp	;lo byte of address
PA 2360	bne	dloop		EG 3280 quit	pla		;pull return adr.	DD 4200	ldy	#\$>temp	;hi byte of address
EM 2370	jsr	untalk	;free serial port	CG 3290	pla			JG 4210	jsr	facmem	;place in temp
CM 2380 ;				PK 3300 cdone	jmp	chrget	;set flags & rts	EC 4220	jsr	chkcom	;commaprint
IC 2390	dex		;forget the <cr>	EG 3310 ;				OC 4230	jsr	frmnum	;get 2nd parm
OB 2400	txa		;put length in .a	AD 3320 ;hex/bin subroutines				NK 4240	jsr	chrhrt	;closing parenprint
NH 2410	sta	len	;save it	IH 3330 ;				AB 4250 ;			
FC 2420	jsr	\$b47d	;reserve space for string	MF 3340 zero	=	*	;set fac to zero	DO 4260	ldx	#\$<modlus	;get adr of modlus
2430	ldy	len	;use length for index	LN 3350	lda	#\$00	;here's the zero	EF 4270	ldy	#\$>modlus	;in .x/.y
OP 2440 ;				DP 3360	ldx	#\$05	;5 bytes + sign	JB 4280	jsr	facmem	;store fac at modlus
GM 2450 dloop2	=	*	;copy string for basic	AK 3370 ;				ID 4290 ;			
LB 2460	lda	wbuf,y	;get byte of string	EP 3380 zilch	sta	exp,x	;zero out byte	AE 4300	lda	#\$<temp	;adr of 1st parm (lo)
HE 2470	sta	(\$62),y	;put in string mem.	DL 3390	dex		;bump index down	GJ 4310	ldy	#\$>temp	;adr of 1st parm (hi)
CP 2480	dey		;bump pointer down	DC 3400	bpl	zilch	;counter roll overprint	DH 4320	jsr	mdiv	;fac = temp/fac
HB 2490	bpl	dloop2		OD 3410	rts			OL 4330	jsr	facarg	;arg = fac
BP 2500	jmp	\$b4ca	;put descrpt on stack	CN 3420 ;				CL 4340	jsr	facint	;fac = int(fac)
EE 2510 ;				DP 3430 convrt	=	*	;ascii digit to true value	EH 4350 ;			
OE 2520 ;				CG 3440	bcc	digit	;chrget flag/digitprint	LE 4360 ;check flag, if div function			
GB 2530 ;@(row,col) function - plot				AJ 3450	ldx	flag	;hex or binaryprint	PI 4370 ;then done, else continue.			
MI 2540 ;cursor and return null string				EJ 3460	bne	chkerr	;binary non-digit	CJ 4380 ;			
MG 2550 ;				GM 3470	cmp	#'a'	;check lower limit	LE 4390	lda	flag	
IC 2560 chkflt	=	\$aefa	;check left paren	HN 3480	bcc	quit	;less than 'a'	MP 4400	beq	done	
KH 2570 chrhrt	=	\$aef7	;check right paren	EA 3490	cmp	#'g'	;check upper limit	AL 4410 ;			
LI 2580 chkcom	=	\$aefd	;check on comma	MF 3500	bcs	chkerr	;bigger than 'f'	JJ 4420	lda	exp	;must have exp in .a
GE 2590 getbyt	=	\$b79e	;get byte into .x	HD 3510	sec			KJ 4430	jsr	subtrt	;fac = arg - fac
HO 2600 plot	=	\$ff0	;plot/fetch cursor	AL 3520	sbc	#\$07	;account for extra 7	OM 4440 ;			
IK 2610 ;				IA 3530 digit	ldx	flag	;hex or binaryprint	BA 4450 ;check flag, if frac function			
JM 2620 xplot	=	*		IG 3540	beq	okay	;hex/any digit is fine	JO 4460 ;then done, else continue.			
DE 2630	jsr	chrget	;get next byte	FE 3550	cmp	#'2'	;bin/check upper limit	MO 4470 ;			
MH 2640	jsr	getprm	;get row/col in x/y	NG 3560	bcs	err2	;bigger than 1	FK 4480	lda	flag	
LJ 2650	cpx	#\$19	;row less than 25print	PE 3570 okay	sec			OE 4490	bmi	done	
PO 2660	bcc	chky	;yes/check column	CM 3580	sbc	#\$30	;convert to true value	KA 4500 ;			
DB 2670 bad	jmp	illegal	;no/illegal quant.	CP 3590	rts			BB 4510	lda	#\$<modlus	;get address of the
NE 2680 chky	cpy	#\$28	;col less than 40print	GI 3600 ;				AN 4520	ldy	#\$>modlus	;modulus in .a/.y
CE 2690	bcs	bad	;no/trash it.	GK 3610 chkerr	=	*	;check illegal quant.	PJ 4530	jsr	mmult	;fac = fac * modlus
AH 2700	clc		;just for looks	NH 3620	cmp	#\$41	;compare with 'a'	GJ 4540	jsr	add5	;add .5 for roundoff
CF 2710	jsr	plot	;plot the cursor	NG 3630	bcc	quit	;less than 'a'	BO 4550	jsr	facint	;truncate garbage
AO 2720	lda	#\$00	;set len to zero	MA 3640	cmp	#\$5b	;compare with 'l'	GE 4560 ;			
KD 2730	jsr	\$b47d	;reserve space	DA 3650	bcs	quit	;greater than 'z'	MP 4570 done	jsr	round	;round the fac
ND 2740	jmp	\$b4ca	;put descrpt on stack	NG 3660 err2	jmp	illegal	;illegal quantity	AN 4580	rts		
ED 2750 ;				MM 3670 ;				EG 4590 ;			
MC 2760 getprm	=	*	;get (a,b) into .x/.y	GN 3680 ;				KA 4600 modlus	=	* + 5	
IC 2770	jsr	chkflt	;check open paren	IN 3690 incexp	=	*	;increment exponent	PC 4610 temp	=	* + 5	
BE 2780	jsr	getbyt	;get first parm	HJ 3700	ldx	exp	;get exponent	CI 4620 ;			
KE 2790	stx	len	;save it here	CP 3710	beq	exit	;fac = 0, don't incr.	CP 4630 .end			
JP 2800	jsr	chkcom	;check on comma	CD 3720	pha		;save byte in .a				
CO 2810	jsr	getbyt	;get second byte	EK 3730	ldx	flag	;use flag for offset				

Command Wedge

By Frank E. DiGioia
Athens, Georgia

Modifying BASIC's Commands

Everyone has their own ideas on how the BASIC interpreter should carry out certain commands. We've all caught ourselves thinking, "If I had written this interpreter, I would have done thus and so. . .". The fact of the matter is that whoever DID write the BASIC interpreter didn't write it just for you and me. They designed the interpreter anticipating what the needs of the AVERAGE user would be. Unfortunately, it was written some years back, when they expected the average user to own a tape drive and not be a particularly sophisticated programmer. Well, times have changed, and thus it seems only fitting that in this issue, which is dedicated to the ROM routines, we should discuss how to modify the existing BASIC commands in order to create a version of BASIC which is perfectly customized to OUR needs today.

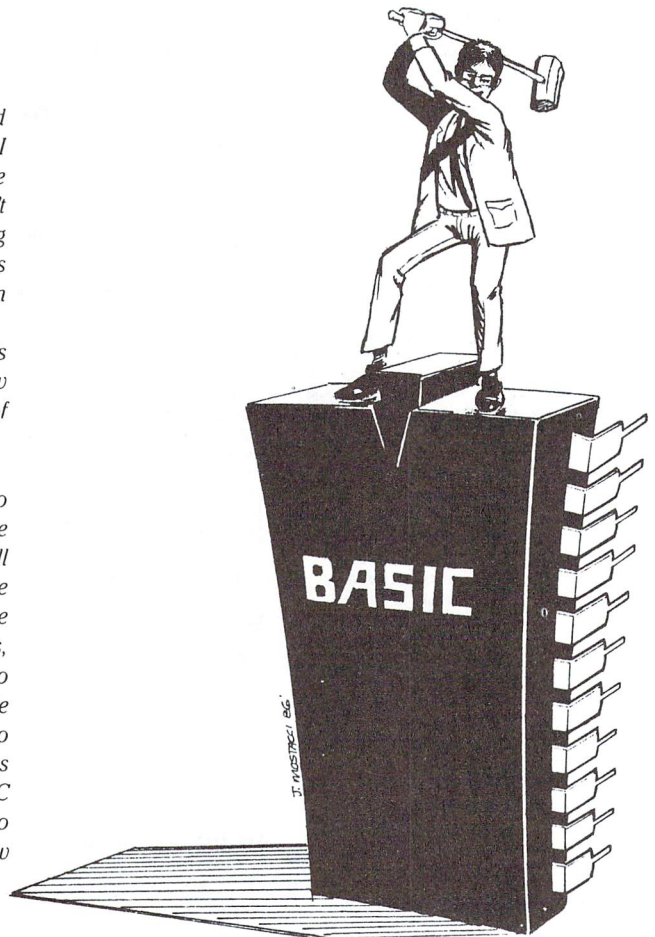
In the previous article we explored a new wedge which allowed us to easily add functions to BASIC. The vector that we used for that wedge is named IEVAL and is located at \$030A/\$030B. In this issue, we will be using IEVAL's twin sister, IGONE located at \$0308/\$0309. These two vectors are almost identical in purpose, the only difference between them being that IEVAL is used for evaluation of functions, and IGONE is used to execute commands. Therefore, before we go any further, please note that all of the material from last issue's article on adding functions through the vector IEVAL can be applied to adding commands through the vector IGONE. The converse is true as well: Everything that we do in this article to modify existing BASIC commands, through the use of IGONE, can be directly applied to modifying existing BASIC functions through the vector IEVAL. Now that we've got that straight, let us begin our example.

Implementing A Command Wedge

While modifying the built-in commands of the BASIC interpreter may sound like quite a job, let me assure you that it is really quite easy to do. Whenever BASIC wants to execute a command, it JMPs through the vector named IGONE at \$308/\$309. The execution routine calls CHRGET to get the token of the command into the accumulator. Bit 7 of this token is then masked off and the resulting value is multiplied by two. This result is the offset into the address table for the address (minus 1) of the routine to execute. This address is then placed on the stack and a JMP is made to CHRGET to get the next byte of text. The RTS in CHRGET causes the PC to be loaded with the address of the routine to execute since this address was just placed on the stack. (Note: This method is almost exactly the same as the one I used in the parser for our function wedge in the last issue.) Armed with this knowledge, we can easily wedge into the execution routine, get the command token and decide if it is one of the ones that we wish to modify. If it is, then we will place the address of OUR routine on the stack, thus executing the new routine instead of the old one. If it is not one that we wish to change, we will give the token back to BASIC for further processing.

An Example Wedge

As an example of this technique, I have written a wedge which changes the action of several of the most commonly used BASIC



commands. The source code is provided and should be studied in order to gain a full understanding of the technique. Even if you have no interest in learning the techniques described here, you should find this example command wedge most useful in your own program development.

Listing 1 contains a BASIC loader for the wedge. Listing 2 is the source code. Listing 3 is an example program which illustrates the use of some of the modified commands. This short program may give you ideas for your own applications in addition to allowing you to test your copy of the wedge.

The commands that we will modify are:

RESTORE	GOSUB	GOTO	WAIT
LIST	LOAD	SAVE	VERIFY
IF/THEN	INPUT	NEW	

In addition, we will make the apostrophe (') act like the REM command and the English pound sign act like an ELSE statement. It is noteworthy that each modified command will work exactly like the original command when the original syntax is used. This feature is very important in maintaining compatibility with older programs. What follows is a description of the modified commands with an example of each one.

- (1) The first change in our modified BASIC is that the apostrophe is now treated as a REM statement. This provides much neater looking listings.

Ex. 10 'this is a comment

- (2) The RESTORE command originally accepted no parameters and was used to set the data pointers back to the beginning of the program. However, there was no means provided to access a particular element of data when needed. The NEW RESTORE command provides TRUE RANDOM ACCESS to your data statements. It takes from zero to two parameters. If no parms are given, the original RESTORE is used.

Ex. 10 read,a,b,c:data 2,4,6,8,10
20 data "PAM", "PAUL", "KELLY"
30 restore 20:read a\$
40 restore 10,4:read x

The RESTORE command in line 30 sets the data pointer to line 20 and A\$ will be assigned the value "PAM". In line 40, the RESTORE command sets the data pointer to the 4th data element of line 10, so X will be assigned the value 8. Attempting to RESTORE to a line# or data element that doesn't exist will result in an informative error. (Try this with the example program in listing 3)

- (3) The new GOTO and GOSUB commands allow variable expressions as arguments.

Ex. 10 if x>0 then goto x*10
20 draw = 1000:paint = 2000
30 gosub draw:gosub paint

- (4) The new WAIT command does exactly what the name implies. It simply waits until a key is pressed. This command is incredibly handy and as an added bonus, once a key is pressed, the new WAIT command leaves the ASCII value of the key at location 2.

Ex. 10 print "Touch any key to continue"
20 wait:print "ascii value ";peek(2)

- (5) The new INPUT command is called INPUT\$ and, as the name implies, is used to input string variables. What sets this command apart from the OLD INPUT is the fact that INPUT\$ will accept any character including quotes, colons and commas with no problems. This capability is extremely important in writing any kind of data processing program. (How many public domain database and mailer programs have you seen that crash if you try to include a comma or colon in an entry?)

Ex. 10 print "enter data: ";input\$ a\$

- (6) The main point of the new IF command is the fact that it is compatible with the modified commands presented here. Since I had to write this routine anyway, I went ahead and added the capability to execute an ELSE clause in the event that the expression evaluates to false. Rather than using the word ELSE, however, the English pound sign (next to CLEAR/HOME) is used as a 'token' for the word ELSE. Be sure to place a colon before the pound sign. In the example below, <L> represents the english pounds sign.

Ex. if x>10 then print "greater":<L>print "less than"

- (7) The new LIST command is exactly like the old, except that it can be used in a program. The old list, you will recall, always returns to direct mode when finished. The new LIST command is invoked by adding an exclamation point after the word LIST. In order to preserve the integrity of the stack, the STOP key is not honored by the new LIST command.

Ex. open4,4:cmd4:list!:print#4:close4

You could use a FOR/NEXT loop to list multiple copies of your program (to give to the members of your user's group, etc.). Another way to use this new capability would be in writing a programming tutorial. You could list the lines before executing them. For example:

10 print "this code moves the sprite"
20 list!1000-1030:wait:gosub1000

- (8) LOAD/SAVE/VERIFY -- A radical change for these three commands! Not only do you no longer need to specify the device number, you don't even need to specify a name! Suppose you LOAD "STAT1" and after modifying the program, you want to save the updated copy. Just type SAVE (with no name) and the program will automatically be replaced for you. But this is not a simple scratch & save. The new SAVE first backs up the old file and THEN replaces it. The backup filename will be the first two letters of the original filename followed by the suffix ".BAK".

Ex. Suppose we load STAT1, then we modify it and type SAVE. It will create a file named ST.BAK which is the OLD STAT1 and then it will replace the old STAT1 with the new, updated copy. Therefore, you can safely and confidently type SAVE when you finish modifying a program knowing that a backup will be made in case the modifications are wrong. The program can then be verified by just typing the word VERIFY. (Note: Only one quote is required when a filename is used with any of these commands)

Ex. (All of these are legal)

load "\$"
load "stat"
load "stat"
load "stat",8,1
save:REM save & replace
save "name":rem save (no replace)
etc.

If you type LOAD, SAVE, or VERIFY with no filename and no name has yet been defined, you'll get a 'MISSING NAME' error (Unless you have Epyx FASTLOAD in place -- it defaults to a filename of M or something on LOAD). Note: Once a name is defined by using it in either LOAD, SAVE or VERIFY, this name will be the default for all three commands until a new one is defined. While LOAD & VERIFY don't mind a filename with a * in it, SAVE is unable to backup such a file and will abort.

Also note: A program can be LOADED and RUN by depressing the SHIFTED RUN/STOP key.

- (9) The NEW command has been modified so that it clears the default name when you type NEW so that you don't accidentally try to save an empty workspace.

Final Notes

The above is just a glimpse of what you can do with the vector IGONE. In addition to just modifying commands, this vector is one of the best places to ADD commands to BASIC. Don't forget, you can add COMMANDS through this vector using exactly the same technique that we used to add FUNCTIONS to BASIC through IEVAL in the last issue. (Likewise, you can modify existing BASIC functions through the vector IEVAL just as we modified existing BASIC commands through the vector IGONE in this issue.)

Up until now, we have been examining wedges which make our lives as programmers a bit easier. In my next article, we will examine a very useful, though somewhat unusual, special purpose wedge which makes USING a program easier. Until then, I will leave you with the following simple projects:

- (1) Add a command or a function (your choice) called !NAME which either prints or returns the name of the file currently in memory. Use the article from last issue as a guide.
- (2) Add a command called !SEND which will send any command to the disk drive. For example !SEND "S0:DATA " will scratch the file named DATA.
- (3) Modify the LIST command so that if you type LIST#n it will do a listing to the previously opened file number n. For example:

```
OPEN 1,8,1,"PROG.L":LIST#1:CLOSE1
```

Hints:

- (1) You can do this without a hint.
- (2) See last issue how to install new commands then just store the DOS command string in a place named CMD and call SEND (see listing 2) as a subroutine.
- (3) This code at the beginning of the routine should do it for you:

```
cmp # '#'      ;number sign?
bne wherever  ;no/do whatever
jsr $b79b     ;get file# in .x
jsr $ffc9     ;set output device
etc. . .     ;same as list!
```

As was the case with our function wedge, this wedge is immune to RUN/STOP-RESTORE and is compatible with most utilities including the DOS Wedge, the Function Wedge and Epyx Fastload Cartridge.

Command Wedge BASIC Loader

FK	100 rem basic loader for command wedge
MK	101 rem by frank e. digioia 12/18/85
EM	102 rem sys 49664 to activate
LN	103 :
CA	104 for adr = 49664 to 50356:read ml
KD	105 cs = cs + ml:poke adr,ml:next
NG	106 ifcs<>85465thenprint "error in data"
PN	107 :
KK	108 data 169, 11, 141, 8, 3, 169, 194, 141
IK	109 data 9, 3, 96, 32, 115, 0, 32, 23
EG	110 data 194, 76, 174, 167, 76, 59, 169, 201
IK	111 data 39, 240, 249, 201, 92, 240, 245, 170
AK	112 data 16, 32, 162, 0, 141, 72, 194, 189
IP	113 data 73, 194, 240, 22, 205, 72, 194, 240
NL	114 data 3, 232, 208, 243, 138, 10, 170, 189

AN	115 data 86, 194, 72, 189, 85, 194, 72, 76
AH	116 data 115, 0, 32, 121, 0, 76, 237, 167
DF	117 data 0, 133, 139, 140, 137, 141, 155, 146
JA	118 data 147, 148, 149, 162, 0, 149, 195, 90
OK	119 data 195, 106, 194, 4, 195, 13, 195, 39
PJ	120 data 195, 247, 195, 7, 196, 14, 196, 4
AN	121 data 196, 249, 194, 208, 3, 76, 29, 168
CO	122 data 32, 212, 194, 165, 95, 164, 96, 56
HG	123 data 233, 1, 176, 1, 136, 133, 65, 132
OI	124 data 66, 32, 121, 0, 240, 50, 32, 253
AO	125 data 174, 32, 158, 183, 138, 240, 41, 202
EL	126 data 240, 38, 160, 4, 177, 65, 201, 131
CP	127 data 208, 31, 200, 177, 65, 240, 44, 201
BH	128 data 58, 240, 40, 201, 34, 240, 21, 201
KA	129 data 44, 208, 239, 202, 208, 236, 152, 24
HI	130 data 101, 65, 133, 65, 144, 2, 230, 66
CE	131 data 96, 169, 131, 44, 169, 34, 133, 251
CB	132 data 200, 177, 65, 240, 6, 197, 251, 240
LJ	133 data 209, 208, 245, 169, 227, 133, 34, 169
BN	134 data 194, 76, 69, 164, 32, 138, 173, 32
CL	135 data 247, 183, 32, 19, 166, 176, 3, 76
HB	136 data 227, 168, 96, 68, 65, 84, 65, 32
CD	137 data 69, 76, 69, 77, 69, 78, 84, 32
IB	138 data 78, 79, 84, 32, 70, 79, 85, 78
DO	139 data 68, 128, 169, 0, 141, 170, 196, 32
DK	140 data 121, 0, 76, 66, 166, 32, 138, 173
AP	141 data 32, 247, 183, 76, 163, 168, 169, 3
PM	142 data 32, 251, 163, 165, 123, 72, 165, 122
PN	143 data 72, 165, 58, 72, 165, 57, 72, 169
EO	144 data 141, 72, 32, 5, 195, 76, 174, 167
AA	145 data 201, 33, 240, 6, 32, 121, 0, 76
GN	146 data 156, 166, 169, 52, 141, 20, 3, 169
JG	147 data 71, 141, 0, 3, 169, 195, 141, 1
NJ	148 data 3, 32, 115, 0, 32, 156, 166, 169
KK	149 data 139, 141, 0, 3, 169, 227, 141, 1
NA	150 data 3, 169, 49, 141, 20, 3, 169, 13
GC	151 data 76, 210, 255, 32, 158, 173, 32, 121
NF	152 data 0, 201, 137, 240, 5, 169, 167, 32
BG	153 data 255, 174, 165, 97, 208, 11, 162, 92
GL	154 data 32, 11, 169, 170, 208, 24, 76, 251
FM	155 data 168, 32, 121, 0, 176, 3, 76, 160
DD	156 data 168, 165, 122, 56, 233, 1, 133, 122
IH	157 data 176, 2, 198, 123, 160, 0, 32, 251
DH	158 data 168, 104, 104, 108, 8, 3, 201, 36
BJ	159 data 240, 3, 76, 191, 171, 32, 115, 0
ML	160 data 240, 15, 162, 0, 32, 207, 255, 201
BJ	161 data 13, 240, 32, 157, 60, 3, 232, 208
GE	162 data 243, 169, 186, 133, 34, 169, 195, 76
IK	163 data 69, 164, 77, 73, 83, 83, 73, 78
NP	164 data 71, 32, 86, 65, 82, 73, 65, 66
MO	165 data 76, 69, 128, 142, 170, 196, 32, 86
JH	166 data 195, 32, 139, 176, 133, 73, 132, 74
LB	167 data 32, 163, 182, 173, 170, 196, 32, 117
OO	168 data 180, 160, 2, 185, 97, 0, 145, 73
LB	169 data 136, 16, 248, 172, 170, 196, 136, 185
GC	170 data 60, 3, 145, 98, 136, 16, 248, 96
FN	171 data 240, 3, 76, 45, 184, 32, 228, 255
AN	172 data 240, 251, 133, 2, 96, 169, 1, 44
ML	173 data 169, 0, 133, 10, 169, 0, 44, 169
LL	174 data 1, 141, 197, 196, 169, 0, 32, 189
FD	175 data 255, 162, 8, 32, 219, 225, 165, 183
NL	176 data 240, 19, 141, 170, 196, 168, 169, 0
NC	177 data 153, 181, 196, 136, 177, 187, 153, 181
DA	178 data 196, 208, 248, 240, 38, 173, 170, 196

LD	179 data 240, 33, 173, 197, 196, 240, 18, 173
LE	180 data 181, 196, 141, 174, 196, 173, 182, 196
NB	181 data 141, 175, 196, 32, 114, 196, 32, 127
IE	182 data 196, 173, 170, 196, 162, 181, 160, 196
CP	183 data 32, 189, 255, 173, 197, 196, 208, 3
OH	184 data 76, 111, 225, 166, 45, 164, 46, 169
NM	185 data 43, 32, 216, 255, 144, 3, 76, 249
BC	186 data 224, 96, 169, 83, 141, 171, 196, 169
PA	187 data 0, 141, 180, 196, 76, 140, 196, 169
EC	188 data 82, 141, 171, 196, 169, 61, 141, 180
NL	189 data 196, 76, 140, 196, 169, 8, 133, 186
FF	190 data 32, 177, 255, 169, 111, 133, 185, 32
AO	191 data 147, 255, 162, 0, 189, 171, 196, 240
HN	192 data 6, 32, 168, 255, 232, 208, 245, 76
KM	193 data 174, 255, 0, 83, 48, 58, 0, 0
DE	194 data 46, 66, 65, 75, 0

Command Wedge Demo Program

```

IF 10 '
OF 20 ' command wedge demo
MG 30 '
NK 40 ' by frank e. digioia
BO 50 ' 11/17/85
KI 60 '
EJ 70 '
AC 80 print "press any key to start":wait
NL 90 print:print "key found! ascii" peek(2)
CE 100 print "q touch any key to test list"
CF 110 wait:list::print "q list done!"
CE 120 print "q choose a subroutine 1, 2, 3"
JK 130 wait:subr = peek(2)-asc("0")
ED 135 if subr<1 or subr>3 then goto 120
LD 140 gosub subr*300+50:'computed gosub
AD 142 print:print "type any chars: ";input$ z$:print "you typed: " z$
GB 150 print:input "goto 170, 180 or 190":a
IB 155 if a<>170 and a<>180 and a<>190 then 150
FH 160 goto a
FN 170 print "q line #170":goto 200
CO 180 print "q line #180":goto 200
IL 190 print "q line #190"
AO 200 print "qq touch any key for restore demo"
AA 210 wait:data 1,2,3,4,5,6,7,8,9,10
HP 220 print "qq data in line 210 printed backwards":print
LE 230 for i = 10 to 1 step -1:restore 210,i:reada:printa::next:print
JH 240 data "georgia", "clemson", "usc"
DH 250 data "colons::", "commas,,", "dot."
CG 260 data "c64", "1541", "mps801"
GP 270 print:print "choose a data line:"
LN 280 input "240, 250 or 260":line
EA 285 if li<>240 and li<>250 and li<>260 then 270
HH 290 print:print "choose a data element:"
EM 300 input "1, 2 or 3":de:if de<1 then 300
ID 310 restore line,de:read a$
AM 320 print "q element is: " a$:goto 270
BG 350 print "executing subr #1":return
PI 650 print "executing subr #2":return
NL 950 print "executing subr #3":return

```

Command Wedge Source Code

```

1000 ;
1010 ;command wedge
1020 ;by frank e. digioia
1030 ;11/17/85
1040 ;
1050 * = $c200
1060 ;
1070 init lda #<cwedge ;install wedge
1080 sta $0308
1090 lda #>cwedge
1100 sta $0309
1110 rts
1120 ;
1130 cwedge = * ;this is the wedge
1140 jsr chrget ;get next byte
1150 jsr chktok ;what is it?
1160 jmp $a7ae ;interpreter loop
1170 ;
1180 rem jmp $a93b ;basic rem command
1190 ;
1200 chktok cmp #$27 ;single quote?
1210 beq rem ;new rem command
1220 cmp #$5c ;'else' pseudo-token
1230 beq rem ;handle as rem
1240 tax ;set flags
1250 bpl wexit ;not a token
1260 ;
1270 ldx #$00 ;use .x as index
1280 sta token ;save for compare
1290 tloop lda toktab,x ;byte from table
1300 beq wexit ;end of table
1310 cmp token ;a match?
1320 beq exec ;yes/execute it
1330 inx ;no/bump index
1340 bne tloop ;keep looking
1350 ;
1360 exec txa ;put offset in .a
1370 asl a ;mult by two
1380 tax ;use as index
1390 lda newadr + 1,x ;put address
1400 pha ;of new routine
1410 lda newadr,x ;on stack.
1420 pha
1430 jmp chrget ;next byte & rts
1440 ;
1450 wexit jsr chrgot ;get byte again
1460 jmp $a7ed ;give it to basic
1470 ;
1480 token .byte $00
1490 ;
1500 toktab .byte $85,$8b,$8c,$89,$8d
1510 .byte $9b,$92,$93,$94,$95,$a2,$00
1520 ;
1530 newadr .word inp-1,if-1,restor-1
1540 .word goto-1,gosub-1,list-1,wait-1
1550 .word load-1,save-1,verfy-1,new-1
1560 ;
1570 ;restore x,y -- all parms optional
1580 ;
1590 chrget = $0073 ;get next byte
1600 chrgot = $0079 ;get last byte
1610 frmmum = $ad8a ;get numeric parm
1620 facint = $b7f7 ;change fac to int
1630 ;
1640 restor = * ;new restore cmd
1650 bne ++5 ;any parms?
1660 jmp $a81d ;no/use rom routine
1670 jsr getprm ;yes/get line & adr
1680 lda $5f ;address lo
1690 ldy $60 ;address hi
1700 sec
1710 sbc #$01 ;subtract 1
1720 bcs ++3 ;decr hi byte?
1730 dey
1740 sta $41 ;data pointer lo
1750 sty $42 ;data pointer hi
1760 jsr chrgot ;another parm?
1770 beq rdone ;no/we're done
1780 ;
1790 jsr $aefd ;yes/check comma
1800 jsr $b79e ;get byte into .x
1810 txa
1820 beq rdone ;0'th element???
1830 dex
1840 beq rdone ;1'st element/done
1850 ldy #$04 ;.y is text index
1860 lda ($41),y ;get byte of text
1870 cmp #$83 ;data statement?
1880 bne findat ;no/find it
1890 ;
1900 loop iny ;comma search loop
1910 lda ($41),y ;get byte from line
1920 beq notfnd ;end of line
1930 cmp #' ;colon?
1940 beq notfnd ;end of data stmtnt
1950 cmp #$22 ;quote?

```


The Transactor	62	July 1986: Volume 7, Issue 01
----------------	----	-------------------------------

Improving The SYS Command

Neil Boyle
Calgary, Alberta

*...make use of those machine language
routines supplied free by Commodore.*

The SYS command in BASIC is very useful – it gives the programmer access to the fast, precise world of machine language. The writers of the Commodore BASIC interpreter realized that programmers often wish to transfer values from BASIC to machine language, so they included the USR command, a specialized form of SYS. Unfortunately, the USR command is limited to transferring one numeric value. A useful extension of the SYS command would allow the passing of multiple parameters in the form of values, variables, equations and strings. A simple method of doing this would be to calculate the values in BASIC and poke them into memory, then SYS to the ML program and have it read the values. Effective, but awkward, slow and clumsy.

A faster and more elegant method is to make use of some of the machine language routines supplied free by Commodore – those in the BASIC interpreter and the KERNAL. There are routines for converting floating point values to integer and back, for evaluating BASIC expressions, for manipulating strings, for printing data in numeric or string form, for storing data in variables, and for printing interpreter or KERNAL error messages. In addition, all mathematical functions handled by the Commodore 64 can be used from a machine language program. These routines are fairly simple to use, and open up innumerable opportunities.

The data which can be passed back and forth between the two languages usually takes one of three forms: string, integer or floating point. Strings are fairly straightforward, and are handled much the same way in each language. Integers, too, are fairly simple, but can be stored in one, two or more bytes. Numbers outside the range of BASIC integers (–32768 to 32767), or those with decimal points, are stored in floating point format, and require 5 or 6 bytes. One advantage of using the interpreter routines rapidly becomes apparent: floating point values can easily be converted to integer and back. Thus, data can be converted from one form to another, manipulated, and converted back, effortlessly (well, almost).

The real problem lies in transferring the parameters from one program to the other. A simple method of doing this using these routines takes the following format:

SYS PA, value1, value2, value3

where PA is the starting address of the ML routine, and value1–3 are the parameters to be passed. For each parameter, ML routines must be called to check for the comma and to evaluate the parameter. The routine at \$AEFD checks for a comma and returns an error message if it is not found. The value following the comma can be evaluated by the routine at \$AD9E. The value can be anything that BASIC can evaluate: strings and string functions, integer or real or boolean equations, variables, etc. If the value is a string or a string function, the result is stored at \$0100, and if numeric, it is stored in floating point format in FAC1. FAC1, or floating point accumulator #1, is located at \$62 to \$65, and is used by the BASIC interpreter for floating point value manipulation. If the type of data to be passed is unknown, reference to two flags will sort this out. The location \$0C is used to indicate the type of data – 255 for string, and 0 for numeric. The type of numeric data is indicated at \$0D – 128 for integer and 0 for floating point. An alternate evaluation routine exists at \$B79E. Following this evaluation, the expression is stored as a one-byte integer (range 0–255) in the X register.

I wrote the following relocatable ML program to demonstrate this method of extending the SYS command. The program is a PRINT AT routine which allows the programmer to specify the column and row of his/her output to the screen. This is simpler and cleaner than fiddling about with embedded cursor controls. The format is:

SYS PA, col, row, value

PA – the address of the start of the ML routine
col – the number of cols from the left screen border (0–39)
row – the number of rows from the top of the screen (0–24)
value – anything the PRINT command can handle

In this example, the SYS command is followed by the parameters to be passed, separated by commas. This line SYS's to PA, the location of the ML routine. The ML program then checks for a comma, evaluates the next parameter, col, and stores the value in the X register. If it is within the acceptable range, it is stored on the stack. The row parameter is placed in the X register in the same manner, and checked for size. The col parameter is pulled from the stack, transferred to the Y register, and the KERNAL Plot routine is used to relocate the cursor. Should either parameter be out of range, the error message "ILLEGAL QUANTITY" is printed.

```
jsr $aefd ;check for comma after SYS
jsr $b79e ;evaluate expression for column number
           (col), store in .X
cpx #$28 ;must be less than 40 ($28)
bcs error ;if not, print error message
txa
pha ;store column value on stack
jsr $aefd ;check for comma after col
jsr $b79e ;evaluate expression for row number (row),
           store in .X
cpx #$19 ;must be less than 25 ($19)
bcs error ;if not, print error message
pla ;retrieve col from stack and
tya ;store in .Y
clc ;clear carry for KERNAL plot routine
jsr $fff0 ;Kernal plot - put cursor at location speci-
           fied in .X and .Y
jsr $0073 ;setup for BASIC interpreter print routine
jmp $aaa0 ;BASIC PRINT routine - print following
           expression
```

```
error ldx #$0e ;error #14 - illegal quantity
```

```
jmp $a437 ;print error specified by value in .X
```

The following BASIC source program will place the program at memory location PA:

AB	100 rem print at - source program
KA	110 pa = 49152: rem location of ml program
AB	120 forj = patopa + 38: reada: pokej, a: next
CB	130 data 32, 253, 174, 32, 158, 183, 224, 40
DM	140 data 176, 24, 138, 72, 32, 253, 174, 32
GO	150 data 158, 183, 224, 25, 176, 12, 104, 168
DK	160 data 24, 32, 240, 255, 32, 115, 0, 76
DG	170 data 160, 170, 162, 14, 76, 55, 164

Possible locations for the ML program are unused Page Two RAM \$02A7 (679), the tape buffer \$033C (828) or free RAM \$C000 (49152). In the program, let PA equal your choice of location and it will be stored there.

One convenient but unusual place to store a short (less than 75 byte) ML program is in a REM statement. To do this,

delete line 110 from the source program, and add the line 10 listed below. Line 10 sets PA equal to the memory address of the first of the 39 spaces in the REM statement and line 120 stores the ML program in the REM statement. RUN the program, then use line 10 as the first line of your program. The remaining lines can be erased, and line 10 can be renumbered, used and stored as wished, but it must remain the first line in the program if PA is kept as location 2063.

```
10 pa = 2063:rem " " <39 spaces>
```

Example: sys pa,4,6, " * ";pa:sqr(144) + 12*6

This example will print an asterisk in column 4 row 6, followed by the value stored in PA (starting location of the ML program), and then the value of the equation (84).

Following is a short list of some of the more useful data manipulation routines in the BASIC interpreter and KERNAL:

Routines used in passing parameters:

\$aefd checks for a comma in the BASIC statement.
 \$ad9e evaluates any expression in the BASIC statement and, if numeric, leaves the results in FAC1. If the expression is a string, it is stored starting at \$0100, and ends with a zero.
 \$b79e evaluates the expression in the BASIC statement, stores the value in FAC1, then converts FAC1 into an integer in the range 0 to 255, and stores the result in the X register.

Routines to convert floating point values in FAC1 to integer values:

\$bc9b converts a floating point value in FAC1 to a four-byte integer in FAC1.
 \$b1bf converts a floating point value in FAC1 to a fixed point integer stored in \$64 and \$65, range -32768 to 32767.
 \$b7a1 converts a floating point value in FAC1 to a fixed point integer in the X register, range 0 to 255.
 \$b1aa converts a floating point value in FAC1 to a 2-byte integer leaving the high byte in the accumulator and the low byte in the Y register.

Routines to convert integer values to floating point values in FAC1:

\$bc44 converts a 2-byte integer in \$62 and \$63 to a floating point value in FAC1.
 \$bc3c converts the accumulator to a floating point value in FAC1.
 \$b3a2 converts the Y register to a floating point value in FAC1.
 \$b391 converts a 2-byte integer, high byte in the accumulator and low byte in the Y register, to a floating point value

in FAC1.

Routine to convert a floating point value in FAC1 to an ASCII string:

\$bddd converts a floating point value in FAC1 to an ASCII string starting at \$0100.

Other useful routines:

\$a437 prints the error message (from the table at \$A19E) corresponding to the value in the X register. For example, loading a 14 in the X register and then jumping to this routine produces the error message "ILLEGAL QUANTITY".

\$aaa0 PRINT command – prints whatever follows, checking for TAB, SPC, commas and semicolons. A jsr to the CHRGET routine is needed before jumping to this routine.

Useful routines and flags from Zero Page and the KERNAL:

\$73 – CHRGET – gets the next character in a BASIC statement.

\$0c – flag: type of data. A value of 255 indicates a string, and a zero indicates numeric data.

\$0d – flag: type of numeric data. A value of 128 indicates an integer, and a zero indicates a floating point value.

\$fff0 – KERNAL plot routine – if the carry flag is cleared, the cursor is placed at the column in the X register and the row in the Y register.

Reference locations:

FAC1 –\$62–\$65 (floating point accumulator)

FAC2 –\$69–\$6E

accumulator –\$30c (780) (.A)

X register –\$30d (781) (.X)

Y register –\$30e (782) (.Y)

Using these BASIC interpreter routines opens many possibilities in combining BASIC and ML programs. All forms of BASIC data, equations, and variables can be passed to ML programs, and ML data can easily be passed back. For a more complete description of these routines, I refer the reader to "Compute!s VIC-20 and Commodore 64 Tool KIT: BASIC" by Dan Heeb, which has been the source of innumerable ideas for me.

Thanks also to Sheldon Leemon and his invaluable book, "Mapping the Commodore 64", for descriptions of these routines. For those more interested in the actual code for these routines, it can be found in "The Anatomy of the Commodore 64" from Abacus Software.

OM	100	rem	printer version	
GD	120	open4,4		
GC	130	print#4,chr\$(27)"p"chr\$(66)		
IG	140	close4		
CE	150	open4,4,2		
BH	160	sys700		
KH	170	.opt oo,p4		
KC	180	;		
CG	190	;ml print at		
OD	200	;		
IE	210	;		
KG	220	chrget = \$73		;get next character
OC	230	errprt = \$a437		;print error type .x
GH	240	print = \$aaa0		;basic print
KD	250	comchk= \$aefd		;check for comma
JK	260	evalxr = \$b79e		;put exp in .x 0–255
GL	270	setcrs = \$fff0		;kernal–place cursor
JA	280	* = \$c000		
IJ	290	;		
OC	300	jsr comchk		;check for comma
OC	310	jsr evalxr		;evaluate col in .x
LN	320	cpx col		
GE	330	bcs error		;branch if col >= 40
IA	340	txa		
HO	350	pha		;store col on stack
KG	360	jsr comchk		;check for comma
ML	370	jsr evalxr		;evaluate row in .x
HE	380	cpx row		
NL	390	bcs error		;branch if row >= 25
AH	400	pla		;get col from stack
JG	410	tay		
KB	420	clc		
PF	430	jsr setcrs		;set cursor at x,y
EH	440	jsr chrget		;first char for print
GB	450	jmp print		
IL	460	rts		
MO	470	error ldx toobig		;parameter too big
GE	480	jmp errprt		;print error .x
AH	490	col .byte 40		;# of columns
DP	500	row .byte 25		;#of rows
HA	510	toobig .byte 14		;illegal quantity

Autoload & the EPROM

by Tom Hughes & Steve McCrystal
Milwaukee, Wisconsin

When the power comes up, so does your application program!

Imagine you're using your Commodore 64 to operate a computerized bulletin board and one stormy day a stray lightning bolt knocks out the local power station. Your BBS crashes. Of course you're not around to pick up the pieces, so when power finally is restored your 64 sits idly, flashing its cursor, while your modem keeps answering and answering and answering those incoming phone calls.

If your 64 had been equipped with Autoload, the computer would have automatically loaded and run the first disk program (your BBS loader) immediately after power was restored. You could have been miles away.

Autoload is a short routine that resides in the 64's KERNAL. Think of Autoload as a "hard" wedge as opposed to the "soft" DOS 5.1 wedge which vanishes as soon as the 64 is turned off. Since it is designed to become a permanent part of the computer's operating system, the only practical way to use Autoload is to burn it into an EPROM (along with the rest of the KERNAL) and then replace your old KERNAL with the Autoload EPROM.

Autoload is able to load and run a disk file because it bypasses the 64's normal start-up or RESET routine. Normally, on power-up or a cold start the 64 jumps to the RESET vector at \$FFFC-FFFF which points to \$FCE2. This routine sets the VIC II chip and the operating system's soft vectors at \$0300, initializes BASIC, resets the stack, and finally turns control of the 64 over to the BASIC interpreter. Autoload performs all of these housekeeping functions, but it also does a LOAD "0:* ",8 and then stuffs the keyboard buffer with the BASIC command RUN. Finally, it jumps to BASIC, which sees it has a RUN command waiting, and that's that.

What if you're not running a BBS? If your 64 isn't a "dedicated" or single-purpose computer, Autoload could become quite a nuisance. Each time you flipped on the 64's power the computer would always - repeat always - try to load and run. To get around this potentially annoying feature, Autoload pauses

about 1 minute before loading, and at any time during this delay you can abort the load by simply pressing the Commodore logo key. This delay also serves a second purpose; it allows a disk drive enough time to reset itself. For some drives that do a self-initialization, like the 4040, there's a chance of a "device not present" error occurring if the drive is accessed too soon.

Nothing's free . . . you'll need to get a suitable EPROM and have access to an EPROM burner, for example, the Promenade (see section below). Also, forget about using a cassette with Autoload because Autoload resides at \$F72C and effectively erases part of a KERNAL cassette routine. However, patches have been placed in the KERNAL which protect you from attempting any cassette operations. One final note: Replacing the KERNAL that came with your 64 with the Autoload custom KERNAL will void your computer's warranty.

If you're willing to part with the use of a cassette, then there's a fair amount of "free space" in the KERNAL for other customizing. For example, the cassette locations between \$F72C-FB8D and \$FB97-FCD0 seem to be ripe territory. Since all this space is available, why not make use of it?

Other KERNAL modifications might include:

- writing your own power-up message at \$E479.
- adding a "hard" DOS wedge.
- an IEEE KERNAL.
- a routine to read and set the time-of-day clocks.
- or any number of short, general purpose programs that you use repeatedly.

Thanks to the bank-switching capabilities of the Commodore 64, custom KERNAL routines can usually be soft tested; that is, run without using an EPROM. The source code for Autoload, for example, includes a conditional assembly variable, called EPROM, that allows Autoload to be soft run "under" the KERNAL ROM itself.

Here are the steps involved for soft running Autoload:

- (1) Assemble Autoload with the variable EPROM = 0.
- (2) Load a machine language monitor and run it.
- (3) Using the monitor, save the KERNAL (\$E000-FFFF) as a disk file. Then load it back. You now have an exact copy of the KERNAL in the RAM under the KERNAL.
- (4) Transfer BASIC to itself. For example, most monitors have a transfer command such as 'T A000 BFFF A000'. This moves BASIC to the RAM under itself.
- (5)* Load the assembled Autoload from disk. This adds Autoload to the RAM KERNAL.
- (6) Create this bank switching routine with the monitor:

```
START SEI
      LDA #$35
      STA $01
      CLI
      RTS
```

This short machine language routine will flip out both the KERNAL and BASIC ROMs when it is called and transfer control of the 64 over to the customized Autoload KERNAL.

- (7) Exit the monitor and do a SYS to START. Autoload is now in place. Next, type SYS 64738 (a RESET) and Autoload should do its stuff.

* NOTE: Commodore's assembler won't allow its object code to be directly assembled to ROM. However, CBM's HILOADER64 and LOLOADER64 programs can be modified to assemble into ROM with a few pokes which place 6502 NOP instructions in a comparison routine:

For LOLOADER64, POKE 2388,234 and POKE 2389,234
For HILOADER64, POKE 51525,234 and POKE 51526,234

Using the Promenade EPROM Programmer

Making a modified KERNAL can be done using any of several EPROM programmers or "burners" available to home users. I use the Promenade sold by Jason-Ranheim Co. of San Jose, California, and recommend it highly because it's inexpensive, simple to operate and very versatile.

Until recently, I used the 2764-type EPROM as a KERNAL replacement chip because of its low cost. However, this is a 28-pin chip. Since the KERNAL ROM has a 24-pin configuration, the 2764 requires an adapter socket and some jumper wires before it can be plugged into the 64's circuit board.

But because of recent price decreases in the Motorola MCM-68764, this chip is now my EPROM of choice. The Motorola EPROM, unlike the 2764, is pin compatible with Commodore's KERNAL chip. The additional cost of the MCM-68764 is offset by not having to fool around with an adapter socket interface.

To program a custom KERNAL with the Promenade, the modified machine code must be loaded into the 64's memory. For example, I relocate the custom KERNAL at \$2000 simply because it's easy to remember. The EPROM is then programmed or "burned" by the EPROM programmer. Using the Promenade (with the Promenade software) with the KERNAL at \$2000 and the 68764 chip, the programming command has the following syntax:

π 8192,16383,0,48,0

"8192" = decimal start address of the code to be burned (\$2000).
"16383" = decimal end address of the code (\$3FFF).
"0" = first byte of the EPROM to be programmed. (Remember, computers start counting from zero).
"48" = Promenade "control word" which tells the burner what type of EPROM it's burning.
"0" = Promenade "program method word" or PMW. This gives the Promenade instructions on how the 68764 should be programmed.

Promenade owners take note: You won't find the above PMW listed in your documentation. I was forced to develop my own PMW because the suggested ones failed to work on the 68764 about 90% of the time.

EPROM burning takes about 4 minutes. If the error light isn't flashing on the Promenade after the burning, then the customized KERNAL is ready to install.

Motorola MCM-68764 EPROMs are available from JAMECO Electronics in the United States as well as other sources. Besides being a direct replacement for the KERNAL, this EPROM can also replace BASIC as well as the 1541 disk drive's ROM.

Autoload Kernal Patch (CBM Assembler format)

```
;put "@:s/kauto"  
.opt nosym  
;*****  
;*                                     *  
;*      autoload kernal               *  
;*      -----                       *  
;*                                     *  
;*      on powerup or reset, loads    *  
;*      the 1st file from drive 0     *  
;*      and then runs the program.   *  
;*      however, a delay period is   *  
;*      provided allowing the user    *  
;*      time to abort the load by     *  
;*      pressing the cmdr logo key.   *  
;*                                     *  
;*      - by tom hughes v240685 -     *  
;*                                     *  
;*****
```



```

.skip      (; sends line feed(s) to printer )
;
;c64 equates
;
basic      =   $0801      ;basic starts here
basini     =   $e3bf      ;initialize basic
basmsg     =   $e422      ;print powerup message
clall      =   $ffe7      ;close all files
close      =   $ffc3      ;close one file
clrchn     =   $ffcc      ;i/o to defaults
dobas      =   $a474      ;basic warm start
keyd       =   $0277      ;keyboard buffer
load       =   $ffd5      ;load ram from disk
ndx        =   $c6        ;# of chars in keybrd buff
setlfs     =   $ffbba     ;set file parameters
setnam     =   $ffbd      ;set file name
settim     =   $ffdb      ;set jiffy clock
shflag     =   $028d      ;shift pattern register
time       =   $a0        ;jiffy clock (3)
vartab     =   $2d        ;basic variable start (2)
vec300     =   $e453      ;set page 3 o.s. vectors
.skip
;
;constants
;
eprom      =   0          ;1 = eprom/0 = soft kernal
wait       =   3          ;wait * 4 = delay in secs
;page 'diversions'
;-----
;cassette routine patches
;-----
.skip
;note: attempted use of a cassette routine
;will result in "illegal device #"
.skip
*          =   $f2ce
        jmp   $f271      ;fix cassette close
*          =   $f38b
        jmp   $f713      ;fix cassette open
*          =   $f539
        jmp   $f713      ;fix cassette load
*          =   $f65f
        jmp   $f713      ;fix cassette save
.skip 2
*          =   $fcef
;-----
;divert system reset
;-----
.skip
.if n eprom <
        stx   $d016      ;reset vicii chip,
        jsr   $fda3      ;initialize i/o,
        jsr   $fd50      ;memory pointers,
        jsr   $fd15      ;soft i/o vectors,
        jsr   $ff5b      ;screen & keyboard
        cli

```

```

        jmp   autold      ;go to autoloader
;page 'autoload'
*          =   $f72c
;-----
;powerup autoloader
;-----
.skip
autold     jsr   vec300    ;set $0300 vectors,
        jsr   basini      ;initialize basic,
        jsr   basmsg      ;print powerup message,
        ldx   #251        ;and reset stack
        txs
        lda   #0          ;zero jiffy clock
        jsr   settim
auto10     lda   shflag
        cmp   #2          ;if cmdr key pressed,
        beq   auto30      ;skip the load
        lda   time + 1
        cmp   #wait       ;else wait till delay is up
        bne   auto10
        jsr   clall       ;then close all files
        lda   #2
        ldx   #8
        ldy   #0          ;ignore file header
        jsr   setlfs
        lda   #3
        ldx   #<filnam
        ldy   #>filnam
        jsr   setnam
        lda   #0          ;load "0:* ",8
        ldx   #<basic
        ldy   #>basic
        jsr   load
        stx   vartab      ;set end-of-basic ptrs
        sty   vartab + 1
        lda   #2          ;close load channel
        jsr   close
        jsr   clrchn
.skip
;autorun routine
;
        ldy   #0
auto20     lda   runit,y    ;write "run" + cr
        sta   keyd,y      ;to keyboard buffer
        iny
        cpy   #4
        bne   auto20
        sty   ndx         ;and set buffer size
auto30     jmp   dobas      ;then run the program
.skip
filnam     .byt   '0:*'
runit      .byt   'run',13
.end

```


SYMASS: A Symbolic Assembler For The Commodore 64

Robert Huehn
Neustadt, Ontario

Now Assemble Any Transactor Program, Anytime!

Symbolic assemblers, used to assemble machine language programs, are essential tools for serious programmers. The merits of machine language need not be discussed here. If you haven't broken down and bought one yet, you've probably been using a monitor such as Supermon. Monitors were never meant for program development. After trying to insert a couple of instructions into a long program with a monitor, you must also readjust the rest of your program properly. Then you think very hard about alternatives.

Unfortunately there were very few choices until now. SYMASS was written to fill the gap. It is a very fast, compact, easy to use assembler with enough features for serious programs. Besides, it's in the public domain. After experiencing SYMASS in action, you will gladly demote your monitor to debugger.

You're likely already familiar with SYMASS syntax, since it is totally compatible with PAL. PAL source code is published often in each issue of The Transactor. SYMASS syntax evolved through many changes from its beginning as a BASIC program (which would take over twenty minutes to assemble early versions.) It now includes most of PAL's features, including the ones most often used in Transactor programs. PAL has no problems assembling SYMASS itself, but SYMASS is faster. SYMASS source code is about 18 K bytes long and PAL takes about 17 seconds to assemble it. SYMASS assembles itself in six seconds.

Type in SYMASS 3.0.GEN, then run it. (It's not long, but you might consider getting the Transactor disk for this issue, especially if you want the source code.) It will create the final program, SYMASS 3.0, on disk. (The generator program could also be modified for tape, since SYMASS doesn't use the disk drive.) The SYMASS 3.0 loader will relocate itself at the top of memory when it is run. Source code is entered with the BASIC editor; use 'sys 700' alone on the first line to call SYMASS. Leave out the PAL's .opt xx statement since SYMASS assembles to memory only. Type 'run' and save the object code with a monitor.

Probably the major limitation of SYMASS is both source and the resulting object code must reside in memory along with SYMASS. SYMASS doesn't take up much room, (about 2.6 K) but you will have problems if the source is too long to fit with the object code.

A partial list of SYMASS/PAL compatible features follows:

* = \$c000	;define start of program
name = \$ff	;assign a value to a symbol
* = * + n	;skip n bytes for storage
;	;comments follow
\$	;hexadecimal value, default is decimal
%	;binary value
'	;ASCII value of character
!	;force absolute addressing
>high, <low	;low or high byte of word
+, -	;add, subtract
.byte \$ff	;store bytes
.word \$ffff	;store words
.asc "text"	;store string of characters
.end	;end assembly

You can use SYMASS without knowing how it works, but the explanation will help you get the most out of it.

SYMASS itself is composed of small modules, each performing a specific function. In general, each module could be replaced by another section of code, if it performs the function correctly. This makes it easier to modify small sections without any side effects. SYMASS was debugged that way.

SYMASS makes two passes over the source code. During the first pass, SYMASS builds a symbol table of all the symbols which appear in the program. It then stores the object code to memory on the second pass, after all unknown symbol values have already been defined. A variable called FLAG is set to 0 or 1, depending which pass SYMASS is currently on.

The source code has already been tokenized by the BASIC editor, but this causes no problems. It even reduces the amount of memory needed for the source. Opcodes such as 'and' are already stored as tokens internally, as are custom pseudo-ops like 'end'.

WORD is the most basic routine to find the next word. WORD defines a word as a sequence of characters ending with a space, colon, semi-colon, or equal sign. It also ignores leading spaces, and has a quote mode that accepts any character except the end of line.

A pointer, AD, and the .y register is always used to access the source code. When WORD is called, the pointer AD is advanced over leading spaces, then the .y register is advanced to the end of the word and the result is stored in LEN. Therefore, LEN is the length of the word, '(ad),y' gives the stop character when .y equals LEN, and the first character when .y equals zero. Two routines, NEXTWORD and NEWWORD, set up AD and call WORD. NEXTWORD starts at the current stop character, so will only get another word if a space separates them. NEWWORD, on the other hand, skips over the stop, and is used to get the expression after an equals sign.

It's important to understand how those routines work if you wish to use them in your own additions to SYMASS.

SYMASS creates a symbol table which starts at the top of memory and grows downward to the end of the source code. A symbol table overflow results if not enough memory is available. Each entry takes ten bytes; eight to store the name, and another two for the value. If tokens are embedded in the name, its actual length could be longer than eight characters, but it's not a good idea.

CRSYM creates a symbol table entry. It decides if there is enough room, then copies the current word into the table. It is your responsibility to make sure a symbol isn't defined twice. Whenever the value of a symbol is needed, FINDSYM is called. FINDSYM returns with the value in the .a and .x registers, or prints an 'undefined symbol' message.

FINDSYM uses the simplest possible search method, searching from beginning to end. It might be worthwhile to use a different method, such as a hash function, to save time. (Calculate the storage address with a special function, such as the remainder of table size / ASCII sum of name.)

The opcode table makes up 728 bytes of SYMASS. Again, FINDOP does a linear search. The more commonly used opcodes are close to the beginning. You could fine-tune the table to your style by counting the number of times each opcode appears in your programs, then rearranging the table in that order. If you do so, change the brk op# and bit op# in DOOP and PUTOP to their new positions. You could also easily add extra opcodes such as skb (skip byte) to the table, changing NOPS to reflect the change.

Two other major routines are EVAL and PUTOP.

EVAL takes the current word, an expression containing no spaces, evaluates it, and returns the result. It can add or subtract symbols, decimal, and hexadecimal numbers. A character enclosed in single quotes will return its ASCII value. A > or < can be placed at the beginning of the expression to return either the high or low byte of the result. The number conversion routines only convert from BASIC's format as a string of characters to a useful two-byte binary number, not both ways. This is why SYMASS gives the end of assembly as a decimal number instead of hex. The BASIC ROM routine that prints 'in xxxxx' is used.

During the first pass DOOP keep track of the current object address with a pointer called PTR. PUTOP is used on the second pass to store the machine code into memory. It recognizes all addressing modes. Since there is no difference in syntax between zero and absolute modes, the correct mode may sometimes be ambiguous.

Suppose you are storing variables in memory after the end of your program, with a label to identify the location. On the first pass, an instruction such as 'lda variable' would normally cause FINDSYM to give an undefined symbol error. FINDSYM therefore tries to guess your meaning by returning the value of PTR for undefined symbols on the first pass. Other assemblers may use zero, and cause an instruction like 'lda variable+1' to produce a phase error. A phase error results when the assembler makes the wrong guess, and reserves an incorrect number of bytes for an instruction.

SYMASS doesn't have phase errors. You can force SYMASS to use absolute mode with a ! prefix, or to zero page by a <, which works by returning the low byte.

You can add your own specialized commands to SYMASS by adding them to the CUSTOP routine. One such command, 'pad' will add a zero to the object code if the current address is odd. You might use it sometime to make sure a jump table doesn't cross a page boundary.

SYMASS leaves room for optimization; the major goals in its design were simplicity, speed and ease of use. WORD, since it is used so often, is a good candidate. PAL doesn't seem to recognize '=' as the end of a word. If the relevant parts of SYMASS were changed, the check could be taken out of WORD. A useful, but probably more complicated improvement is assembly to disk.

SYMASS's hidden strength is the ease with which it can be modified, compared to commercial programs which do not provide source code. You can also study SYMASS just to learn how to write an assembler. In the end though, SYMASS is a tool which will enable you to write machine language programs as complex as your growing skills allow.

SYMASS Source Listing

The source code for SYMASS 3.1 has not been verified - it's here for reference only - we couldn't imagine anyone entering it by hand. If you want source we suggest you type in the loader and unassemble it, otherwise get Disk #12 for this issue.

```
100 sys700
110: < < < < symass 3.1 > > > >
120: symbolic assembler
130: robert huehn june, october 1985
140: sm3.103
150 * = $c000
160:
170 ;zero page equates
180 stasc = $50 ;start of source
190 stavar = $2d ;end of source
200 memsiz = $37 ;top of symbol table
210 link = $4e ;basic line link
220 line = $39 ;current line number
230 ad = $7a ;current source address
240 symptr = $52 ;symbol value pointer
250 symend = $57 ;bottom of symbol table
260 ptr = $59 ;current object address
270 opptr = $5b ;opcode table pointer
280 len = $5d ;length of word
290 t1 = $26 ;temporary number
300 t2 = $28 ;storage for eval
310 ss = $2a ;sign save
320 i3 = $5e
330 i4 = $5f
340 i5 = $60
350 flag = $02 ;first or second pass
360 constants
370 nops = 55 ;number of instructions
380 ready = $a474 ;basic ready
390 inline = $bdc2 ;print 'in line'
400 contbas = $a7ae ;continue basic
410 list = $a6c9 ;list line
420 findline = $a613 ;find basic line
430:
440 ;main program
450:
460 init = * ;begin first pass
470:
480 lda *0
490 sta flag
500 ldx * < messstar ;start
510 ldy * > messstar
520 jsr printmsg
530 ldx $3a
540 inx
550 bne it1
560 jmp ready ;since in direct mode
570 it1 ldx * < messfir ;first pass
580 ldy * > messfir
590 jsr printmsg
600 lda memsiz ;init symbol table
610 sta symend
620 lda memsiz + 1
630 sta symend + 1
640 inc ad
650 bne it2
660 inc ad + 1
670 it2 lda ad
680 sta stasrc
690 sta link
700 lda ad + 1
710 sta stasrc + 1
720 sta link + 1
730:
740 newline = * ;start next line
750:
760 jsr nextline
770 bne getword
780 jmp secpass
790:
800 getword = * ;process word
810:
820 jsr word
830 bne gw1
840 cmp * $b2 ;= token
850 bne next
860 jmp addval
870 gw1 ldx *0 ;check for * = + + 1
880 lda (ad,x)
890 cmp * $ac ;+ token
900 bne gw2
910 jsr doptr
920 jmp next
930 gw2 lda (ad,y)
940 cmp * $b2 ;=
950 bne gw3
960 jmp addsym
970 gw3 jsr findop
980 bcc gw4
990 jmp doop
1000 gw4 ldy *0
1010 lda (ad,y)
1020 cmp * * *
```

```
1030 bne label
1040 jmp custop
1050:
1060 label = * ;save word, current address
1070:
1080 jsr crsym
1090 ldy *0
1100 lda ptr
1110 sta (symptr),y
1120 iny
1130 lda ptr + 1
1140 sta (symptr),y
1150:
1160 next = * ;get ready for next word
1170:
1180 ldy len
1190 lda (ad),y
1200 cmp * *
1210 beq n1
1220 cmp * *
1230 beq n1
1240 jmp newline
1250 n1 iny
1260 tya
1270 clc
1280 adc ad
1290 sta ad
1300 bcc n
1310 inc ad + 1
1320 n jmp getword
1330:
1340 secpass = * ;begin second pass
1350:
1360 inc flag
1370 ldx * < messsec ;second pass
1380 ldy * > messsec
1390 jsr printmsg
1400 lda stasrc ;put link at start
1410 sta link
1420 lda stasrc + 1
1430 sta link + 1
1440:
1450 newline2 = * ;start next line
1460:
1470 jsr nextline
1480 bne getword2
1490 jmp finish
1500:
1510 getword2 = * ;process word
1520:
1530 jsr word
1540 beq next2
1550 ldx *0
1560 lda (ad,x)
1570 cmp * $ac
1580 bne g2w1
1590 jsr doptr
1600 jmp next2
1610 g2w1 jsr findop
1620 bcc g2w2
1630 jmp putop
1640 g2w2 ldy *0
1650 lda (ad),y
1660 cmp * *
1670 bne next2
1680 jmp custop
1690:
1700 next2 = * ;get ready for next word
1710:
1720 ldy len
1730 lda (ad,y)
1740 cmp * $20
1750 beq n2x1
1760 cmp * $3a
1770 beq n2x1
1780 jmp newline2
1790 n2x1 iny
1800 tya
1810 clc
1820 adc ad
1830 sta ad
1840 bcc n2x
1850 inc ad + 1
1860 n2x jmp getword2
1870:
1880 finish = * ;end
1890:
1900 ldx * < messacc ;assembly complete
1910 ldy * > messacc
1920 jsr printmsg
1930 lda ptr
1940 sta line
1950 lda ptr + 1
1960 sta line + 1
1970 jsr inline
1980 jmp ready
1990:
2000 ;subroutines used by main program
2010:
2020 addsym = * ;save symbol with value
```

```
2030:
2040 jsr crsym
2050 jsr newword
2060 jsr eval
2070 ldy *0
2080 sta (symptr),y
2090 iny
2100 txa
2110 sta (symptr),y
2120 jmp next
2130:
2140 addval = * ;change label into symbol
2150:
2160 jsr newword
2170 jsr eval
2180 ldy *0
2190 sta (symptr),y
2200 iny
2210 txa
2220 sta (symptr),y
2230 jmp next
2240:
2250 crsym = * ;create symbol table entry
2260:
2270 lda symend ;lower symend to
2280 sec ;make room
2290 sbc *10
2300 sta symend
2310 bcs cs1
2320 dec symend + 1
2330 cs1 cmp stavar ;check for
2340 lda symend + 1
2350 sbc stavar + 1
2360 bcs cs2
2370 ldx * < messsto ;symbol table
2380 ldy * > messsto ;overflow
2390 jsr printmsg
2400 jsr inline
2410 jmp listline
2420 cs2 clc ;point symptr to
2430 lda symend ;symbol value address
2440 adc *8
2450 sta symptr
2460 lda symend + 1
2470 adc *0
2480 sta symptr + 1
2490 ldy *8 ;erase space for name
2500 lda *0
2510 cs4 dey
2520 sta (symend),y
2530 bne cs4
2540 ldy len ;max length is 8
2550 cs5 dey ;copy symbol name
2560 lda (ad),y
2570 sta (symend),y
2580 tya
2590 bne cs5
2600 rts
2610:
2620 doop = * ;move ptr past instruction
2630:
2640 ldy *0
2650 lda (ad),y
2660 cmp * *
2670 beq do3
2680 cmp * * b
2690 bne do1
2700 cpx * $21 ;brk op*
2710 beq do1
2720 cpx * $20 ;bit op*
2730 beq do1
2740 jsr nextword
2750 do2 lda *2
2760 bne do
2770 do3 jsr nextword
2780 lda *3
2790 bne do
2800 do1 jsr nextword
2810 bne do2
2820 lda *1
2830 bne do
2840 do2 ldy *0
2850 lda (ad),y
2860 cmp * * *
2870 beq do2
2880 cmp * * (
2890 beq do2
2900 ldy len
2910 dey
2920 beq do5
2930 dey
2940 beq do5
2950 lda (ad),y
2960 cmp * *
2970 bne do5
2980 iny ;recognize intended absolute
2990 lda (ad),y
3000 ldy *7
3010 cmp * * x
3020 beq do7
```

```
3030 iny
3040 do7 lda (opptr),y
3050 cmp * $fa
3060 beq do3
3070 ldy len
3080 dey dey
3090 sty len
3100 jsr eval
3110 inc len
3120 inc len
3130 cpx *0
3140 jmp do6
3150 do5 jsr eval
3160 do6 beq do2
3170 do3 lda *3
3180 do clc
3190 adc ptr
3200 sta ptr
3210 bcc do4
3220 inc ptr + 1
3230 do4 jmp next
3240:
3250 doptr = * ;change ptr eg. * = + + 2
3260:
3270 jsr nextword
3280 jsr newword
3290 jsr eval
3300 sta ptr
3310 stx ptr + 1
3320 rts
3330:
3340 ;eval routines begin here
3350:
3360 literal = * ;return single ascii value
3370:
3380 iny
3390 lda (ad),y
3400 sta t1
3410 lda *0
3420 sta t1 + 1
3430 iny iny
3440 jmp last
3450:
3460 sym = * ;find end and call findsym
3470:
3480 syl iny
3490 cpy len
3500 beq sy
3510 lda (ad),y
3520 cmp * $aa ;+
3530 beq sy
3540 cmp * $ab ;-
3550 bne syl
3560 sy sty t1
3570 jsr findsym
3580 ldy t1
3590 sta t1
3600 stx t1 + 1
3610 jmp last
3620:
3630 eval = * ;evaluate single expression
3640:
3650 lda *0
3660 sta t2
3670 sta t2 + 1
3680 sta ss
3690 sta t4
3700 ev1 ldy *0
3710 lda (ad),y
3720 cmp * $
3730 bne ev8
3740 jmp hex
3750 ev8 cmp * $22 ;'
3760 beq literal
3770 cmp * $ac ;*
3780 beq ptrval
3790 cmp * $b1 ;>
3800 beq hilo
3810 cmp * $b3 ;<
3820 beq hilo
3830 cmp * * % ;%
3840 bne ev9
3850 jmp bin
3860 ev9 sec
3870 sbc * $30
3880 bcc sym
3890 cmp * $0a
3900 bcs sym
3910 jmp deci
3920:
3930 hilo = * ;> or < byte extractions
3940:
3950 sta t4
3960 inc ad
3970 bne hl
3980 inc ad + 1
3990 hl dec len
4000 bne ev1
4010:
4020 ptrval = * ;give current address
```


4030 ;				5030	adc	*0		6030	lda	opptr		7030	ldy	len	
4040	iny			5040	sta	t1+1		6040	clc			7040	tya		
4050	lda	ptr		5050	bcs	iq		6050	adc	*\$0d		7050	clc		
4060	sta	t1		5060	iny			6060	sta	opptr		7060	adc	ad	
4070	lda	ptr+1		5070	bne	hx1		6070	bcc	fo5		7070	sta	ad	
4080	sta	t1+1		5080 hx	jmp	last		6080	inc	opptr+1		7080	bcc	nw	
4090 ;				5090 ;				6090 fo5	cpx	*nops		7090	inc	ad+1	
4100 last	=	*	;perform last sign	5100 iq	=	*	;illegal quantity	6100	bne	fo1		7100 nw	jmp	word	
4110 ;				5110	ldx	*<messiq		6110	clc			7110 ;			
4120	lda	ss		5120	ldy	*>messiq		6120	rts			7120 printmsg	=	*	;print message
4130	bne	ev3		5130	jsr	printmsg		6130 ;				7130 ;			
4140	lda	t1	;no sign	5140	jsr	inline		6140 findsym	=	*	;find symbol, return	7140	stx	t1	
4150	sta	t2		5150	jmp	listline		6150 ;	value in a.x			7150	sty	t1+1	
4160	lda	t1+1		5160 ;				6160 ;				7160	ldy	*0	
4170	sta	t2+1		5170 deci	=	*	;convert decimal	6170	lda	memsiz		7170 pm1	lda	(t1),y	
4180	jmp	sign		5180 ;				6180	sta	symptr		7180	beq	pm	
4190 ev3	cmp	*\$aa	;+	5190	lda	*0		6190	lda	memsiz+1		7190	jsr	\$ffd2	;print character
4200	bne	ev4		5200	sta	t1		6200	sta	symptr+1		7200	iny		
4210	clc			5210	sta	t1+1		6210 fs1	lda	symptr		7210	bne	pm1	
4220	lda	t1		5220 del	lda	(ad),y		6220	sec			7220	pm	rts	
4230	adc	t2		5230	sec			6230	sbc	*10		7230 ;			
4240	sta	t2		5240	sbc	*\$30		6240	sta	symptr		7240 ;	putop routines begin here		
4250	lda	t1+1		5250	bcc	de		6250	bcs	fs2		7250 ;			
4260	adc	t2+1		5260	cmp	*\$0a		6260	dec	symptr+1		7260 relative	=	*	;calculate offset
4270	bcc	sign		5270	bcs	de		6270 fs2	cmp	symend		7270 ;			
4280	jmp	iq		5280	pha			6280	lda	symptr+1		7280	ldy	*3	
4290 ev4	sec	;- (default)		5290	lda	t1		6290	sbc	symend+1		7290	lda	(opptr),y	
4300	lda	t2		5300	ldx	t1+1		6300	bcs	fs3		7300	jsr	putoutop	
4310	sbc	t1		5310	asl	t1		6310	lda	flag		7310	jsr	nextword	
4320	sta	t2		5320	rol	t1+1		6320	bne	fs8		7320	jsr	eval	
4330	lda	t2+1		5330	bcs	iq		6330	lda	ptr	;return ptr on 1st pass	7330	sec		
4340	sbc	t1+1		5340	asl	t1		6340	ldx	ptr+1		7340	sbc	*1	
4350	sta	t2+1		5350	rol	t1+1		6350	rts			7350	bcs	r11	
4360	bcc	iq		5360	bcs	iq		6360 fs8	ldx	*<messus	;undefined symbol	7360	dex		
4370 ;				5370	adc	t1		6370	ldy	*>messus		7370 r11	sec		
4380 sign	=	*	;save sign or stop	5380	sta	t1		6380	jsr	printmsg		7380	sbc	ptr	
4390 ;				5390	txa			6390	jsr	inline		7390	sta	t1	
4400	cpy	len		5400	adc	t1+1		6400	jmp	listline		7400	txa		
4410	beq	ev		5410	sta	t1+1		6410 fs3	ldy	*0		7410	sbc	ptr+1	
4420	lda	(ad),y		5420	bcs	iq		6420 fs4	lda	(symptr),y		7420	tax		
4430	sta	ss		5430	asl	t1		6430	beq	fs7		7430	clc		
4440	iny			5440	rol	t1+1		6440	cmp	(ad),y		7440	lda	t1	
4450	tya			5450	bcs	iq		6450	bne	fs1		7450	adc	*\$80	
4460	clc			5460	pla			6460	iny			7460	txa		
4470	adc	ad		5470	adc	t1		6470	cpy	*8		7470	adc	*0	
4480	sta	ad		5480	sta	t1		6480	bcc	fs4		7480	beq	r1	
4490	bcc	ev5		5490	lda	t1+1		6490 fs7	cpy	t1	;length	7490	ldx	*<messboor	;branch out of
4500	inc	ad+1		5500	adc	*0		6500	bne	fs1		7500	ldy	*>messboor	;range
4510 ev5	sec			5510	sta	t1+1		6510 fs9	ldy	*9	;found it	7510	jsr	printmsg	
4520	lda	len		5520	bcs	iq		6520	lda	(symptr),y		7520	jsr	inline	
4530	sty	len		5530	iny			6530	tax			7530	jmp	listline	
4540	sbc	len		5540	bne	del		6540	dey			7540 r1	lda	t1	
4550	sta	len		5550 de	jmp	last		6550	lda	(symptr),y		7550	jsr	putout	
4560	jmp	ev1		5560 ;				6560	rts			7560	jmp	next2	
4570 ev	lda	t4		5570 bin	=	*	;convert binary	6570 ;				7570 ;			
4580	bne	ev6		5580 ;				6580 listline	=	*	;list offending line	7580 imm	=	*	;do immediate mode *
4590	lda	t2		5590	iny			6590 ;				7590 ;			
4600	ldx	t2+1		5600	lda	*0		6600	lda	line		7600	inc	ad	
4610	rts			5610	sta	t1		6610	sta	\$14		7610	bne	im1	
4620 ev6	cmp	*\$b1	>	5620	sta	t1+1		6620	lda	line+1		7620	inc	ad+1	
4630	bne	ev7		5630 bn1	lda	(ad),y		6630	sta	\$15		7630 im1	ldy	*\$0a	
4640	lda	t2+1		5640	sec			6640	jsr	findline		7640	lda	(opptr),y	
4650	ldx	*0		5650	sbc	*\$30		6650	jsr	list		7650	jsr	putoutop	
4660	rts			5660	bcc	bn		6660	jmp	ready		7660	dec	len	
4670 ev7	lda	t2	<	5670	cmp	*2		6670 ;				7670	jsr	eval	
4680	ldx	*0		5680	bcs	bn		6680 nextline	=	*	;ready for next line	7680	jsr	putout	
4690	rts			5690	asl	t1		6690 ;				7690	jmp	next2	
4700 ;				5700	rol	t1+1		6700	lda	link	;move ad to next line	7700 ;			
4710 hex	=	*	;convert hex number	5710	bcs	iq		6710	sta	ad		7710 indirect	=	*	;do (x) else (y)
4720 ;				5720	adc	t1		6720	lda	link+1		7720 ;			
4730	iny			5730	sta	t1		6730	sta	ad+1		7730	inc	ad	
4740	lda	*0		5740	lda	t1+1		6740	ldy	*0	;new link	7740	bne	ind1	
4750	sta	t1		5750	adc	*0		6750	lda	(ad),y		7750	inc	ad+1	
4760	sta	t1+1		5760	sta	t1+1		6760	sta	link		7760 ind1	lda	len	
4770 hx1	lda	(ad),y		5770	iny			6770	iny			7770	sec		
4780	sec			5780	bne	bn1		6780	lda	(ad),y		7780	sbc	*4	
4790	sbc	*\$30		5790 bn	jmp	last		6790	sta	link+1		7790	tay		
4800	bcc	hx		5800 ;				6800	beq	nl	;end of source return z set	7800	sty	len	
4810	cmp	*\$0a		5810 findop	=	*	;set carry if opcode,	6810	iny		;new line number	7810	lda	(ad),y	
4820	bcc	hx2		5820	opptr points to position,63999			6820	lda	(ad),y		7820	ldy	*11	
4830	sbc	*\$11		5830	;x holds opcode number			6830	sta	line		7830	cmp	*"	
4840	bcc	hx		5840 ;				6840	iny			7840	beq	ind2	
4850	cmp	*\$06		5850	lda	*<optab	;opcode table	6850	lda	(ad),y		7850	iny		
4860	bcs	hx		5860	sta	opptr		6860	sta	line+1		7860 ind2	lda	(opptr),y	
4870	adc	*\$0a		5870	lda	*>optab		6870	clc		;move ad over link and line	7870	jsr	putoutop	
4880 hx2	asl	t1		5880	sta	opptr+1		6880	lda	ad		7880	jsr	eval	
4890	rol	t1+1		5890	ldx	*0		6890	adc	*4		7890	jsr	putout	
4900	bcs	iq		5900 fo1	ldy	*0		6900	sta	ad		7900	inc	len	
4910	asl	t1		5910 fo2	lda	(opptr),y		6910	bcc	nl		7910	inc	len	
4920	rol	t1+1		5920	beq	fo3		6920	inc	ad+1	;return z clear	7920	inc	len	
4930	bcs	iq		5930	cmp	(ad),y		6930 nl	rts			7930	jmp	next2	
4940	asl	t1		5940	bne	fo4		6940 ;				7940 ;			
4950	rol	t1+1		5950	iny			6950 newword	=	*	;get next word past =	7950 putop	=	*	;generates machine code
4960	bcs	iq		5960	cpy	*3		6960 ;				7960 ;			
4970	asl	t1		5970	bcc	fo2		6970	ldy	len		7970	ldy	*0	
4980	rol	t1+1		5980 fo3	cpy	len		6980	iny			7980	lda	(ad),y	
4990	bcs	iq		5990	bne	fo4		6990	.byte	\$2c		7990	cmp	*"	
5000	adc	t1		6000	sec			7000 ;				8000	bne	pop5	
5010	sta	t1		6010	rts			7010 nextword	=	*	;get next word	8010	jmp	jump	
5020	lda	t1+1		6020 fo4	inx			7020 ;				8020 pop5	cmp	*"b"	

8030	bne	pop1		9030	jsr	eval	10030	by1	ldy	*0	11030	:
8040	cpx	*\$21	:brk op*	9040	jsr	putout	10040	by2	lda	(adj.y	11040	optab = * :opcode table
8050	beq	pop1		9050	txa		10050		cmp		11050	:
8060	cpx	*\$20	:bit op*	9060	jsr	putout	10060		beq	by3	11060	asc *lda :.byte \$ad, \$bd, \$69, \$a5, \$b5, \$fa, \$fa, \$a9, \$a1, \$b1
8070	beq	pop1		9070	inc	len	10070	by9	iny		11070	asc *sta :.byte \$8d, \$9d, \$99, \$85, \$95, \$fa, \$fa, \$fa, \$81, \$91
8080	jmp	relative		9080	jmp	next2	10080		cpy	t3	11080	asc *bne :.byte \$d0, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8090	pop1	jsr	nextword	9090	:		10090		bne	by2	11090	asc *beq :.byte \$f0, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8100	bne	pop2		9100	putoutop = *	:verify op mode	10100	by3	lda	flag	11100	asc *cmp :.byte \$cd, \$dd, \$d9, \$c5, \$d5, \$fa, \$fa, \$c9, \$c1, \$d1
8110	ldy	*9		9110	:		10110		beq	by6	11110	asc *jsr :.byte \$20, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8120	lda	(optr).y		9120	cmp	*\$fa	10120		sty	len	11120	asc *ldx :.byte \$ae, \$fa, \$be, \$a6, \$fa, \$b6, \$fa, \$a2, \$fa, \$fa
8130	jsr	putoutop		9130	bne	putout	10130		iny		11130	asc *rts :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8140	jmp	next2		9140	ldx	<*messim :illegal mode	10140		lda	t3	11140	asc *ldy :.byte \$ac, \$bc, \$fa, \$a4, \$b4, \$fa, \$fa, \$a0, \$fa, \$fa
8150	pop2	ldy	*0	9150	ldy	>*messim	10150		sty	t3	11150	asc *bmi :.byte \$30, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8160	lda	(adj.y		9160	jsr	printmsg	10160		sec		11160	asc *dec :.byte \$ce, \$de, \$fa, \$c6, \$d6, \$fa, \$fa, \$fa, \$fa, \$fa
8170	cmp	*\$*		9170	jsr	inline	10170		sbc	t3	11170	byte \$af,0,0,\$2d,\$3d,\$39,\$25,\$35,\$fa,\$fa,\$29,\$21,\$31 :and
8180	bne	pop3		9180	jmp	listline	10180		sta	t3	11180	asc *bcs :.byte \$b0, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8190	jmp	imm		9190	:		10190		bcs	by4	11190	asc *inc :.byte \$ee, \$fa, \$fa, \$e6, \$f6, \$fa, \$fa, \$fa, \$fa, \$fa
8200	pop3	cmp	*\$*	9200	putout = *	:output object code	10200		lda	*0	11200	asc *bcc :.byte \$90, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8210	bne	pop4		9210	:		10210		sta	t3	11210	asc *tya :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$98, \$fa, \$fa
8220	jmp	indirect		9220	ldy	*0	10220	by4	jsr	eval	11220	asc *bpl :.byte \$10, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8230	pop4	cmp	*\$*	9230	sta	(ptr).y	10230		jsr	putout	11230	asc *asl :.byte \$0e, \$1e, \$fa, \$06, \$16, \$fa, \$0a, \$fa, \$fa, \$fa
8240	bne	absolute		9240	inc	ptr	10240		lda	t5	11240	asc *clc :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$18, \$fa, \$fa, \$fa
8250	:forced absolute by ! prefix			9250	bne	pt	10250		beq	by5	11250	asc *adc :.byte \$6d, \$7d, \$79, \$65, \$75, \$fa, \$69, \$61, \$71
8260	inc	ad		9260	inc	ptr + 1	10260		txa		11260	byte \$45,\$b0,0,\$4d,\$5d,\$59,\$45,\$55,\$fa,\$49,\$41,\$51 :eor
8270	bne	fr		9270	pt	rts	10270		jsr	putout	11270	asc *txa :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$8a, \$fa, \$fa, \$fa
8280	inc	ad + 1		9280	:		10280	by5	lda	t3	11280	asc *cpx :.byte \$ec, \$fa, \$fa, \$e4, \$fa, \$fa, \$e0, \$fa, \$fa, \$fa
8290	fr	dec	len	9290	word = *	:basic routine to get word	10290		beq	by	11290	asc *jmp :.byte \$4c, \$6c, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8300	byte	\$2c		9300	(adj) must point to start		10300		ldy	len	11300	asc *tax :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$aa, \$fa, \$fa, \$fa
8310	:			9310	:ignores leading spaces		10310		iny		11310	asc *iny :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$c8, \$fa, \$fa, \$fa
8320	absolute = *	:three byte mode		9320	: = * copied only in quote mode		10320		tya		11320	asc *sty :.byte \$8c, \$fa, \$fa, \$84, \$94, \$fa, \$fa, \$fa, \$fa, \$fa
8330	:			9330	:return .y-length, stop char in .a		10330		clc		11330	byte \$b0,\$41,0,\$0d,\$1d,\$19,\$05,\$15,\$fa,\$09,\$01,\$11 :ora
8340	lda	*0		9340	:		10340		adc	ad	11340	asc *dey :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$88, \$fa, \$fa
8350	sta	t5		9350	ldx	*0	10350		sta	ad	11350	asc *dex :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$ca, \$fa, \$fa, \$fa
8360	ldx	*3		9360	ldy	*0	10360		bcc	by1	11360	asc *stx :.byte \$8e, \$fa, \$fa, \$86, \$fa, \$96, \$fa, \$fa, \$fa, \$fa
8370	ldy	len		9370	lda	(adj).y	10370		inc	ad + 1	11370	asc *sbc :.byte \$ed, \$fd, \$f9, \$e5, \$f5, \$fa, \$e9, \$e1, \$f1
8380	dey			9380	beq	w5	10380		bne	by1	11380	asc *bit :.byte \$2c, \$fa, \$fa, \$24, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8390	beq	abl		9390	cmp	*\$22	10390	by6	clc		11390	asc *brk :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$00, \$fa, \$fa, \$fa
8400	dey			9400	beq	w4	10400		lda	t5	11400	asc *bvc :.byte \$50, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8410	beq	abl		9410	cpx	*\$80	10410		beq	by7	11410	asc *bvs :.byte \$70, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$fa
8420	lda	(adj).y		9420	beq	w2	10420		lda	*1	11420	asc *cid :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$d8, \$fa, \$fa, \$fa
8430	cmp	*\$*		9430	cmp	*\$*	10430	by7	adc	*1	11430	asc *cli :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$58, \$fa, \$fa, \$fa
8440	bne	abl		9440	beq	w5	10440		adc	ptr	11440	asc *clv :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$b8, \$fa, \$fa, \$fa
8450	sty	len		9450	cmp	*\$*	10450		sta	ptr	11450	asc *cpy :.byte \$cc, \$fa, \$fa, \$c4, \$fa, \$fa, \$c0, \$fa, \$fa, \$fa
8460	inx			9460	beq	w5	10460		bcc	by8	11460	asc *inx :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$e8, \$fa, \$fa, \$fa
8470	iny			9470	cmp	*\$b2	10470		inc	ptr + 1	11470	asc *lsr :.byte \$4e, \$5e, \$fa, \$46, \$56, \$fa, \$4a, \$fa, \$fa, \$fa
8480	lda	(adj).y		9480	beq	w5	10480	by8	cpy	t3	11480	asc *nop :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$ea, \$fa, \$fa, \$fa
8490	cmp	*\$x		9490	cmp	*\$*	10490		bne	by9	11490	asc *pha :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$48, \$fa, \$fa, \$fa
8500	beq	abl		9500	beq	w3	10500	by	jmp	cp	11500	asc *php :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$08, \$fa, \$fa, \$fa
8510	inx			9510	iny	:copy	10510	:			11510	asc *pla :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$68, \$fa, \$fa, \$fa
8520	abl	stx	t3	9520	bne	w1	10520	asc	= *	:asc	11520	asc *plp :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$28, \$fa, \$fa, \$fa
8530	jsr	eval		9530	cpy	*0	10530	:			11530	asc *rol :.byte \$2e, \$3e, \$fa, \$26, \$36, \$fa, \$2a, \$fa, \$fa, \$fa
8540	beq	ab2		9540	bne	w5	10540		jsr	nextword	11540	byte \$52,\$b0,0,\$6e,\$7e,\$fa,\$66,\$76,\$fa,\$6a,\$fa,\$fa,\$fa :ror
8550	ab4	ldy	t3	9550	inc	ad	10550		ldy	*1	11550	asc *rti :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$40, \$fa, \$fa, \$fa
8560	lda	(optr).y		9560	bne	w1	10560	as1	lda	(adj).y	11560	asc *sec :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$38, \$fa, \$fa, \$fa
8570	jsr	putoutop		9570	inc	ad + 1	10570		cmp	*\$22	11570	asc *sed :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$f8, \$fa, \$fa, \$fa
8580	lda	t2		9580	bne	w1	10580		beq	as	11580	asc *sei :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$78, \$fa, \$fa, \$fa
8590	jsr	putout		9590	w4	txa :toggle	10590		ldx	flag	11590	asc *tay :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$a8, \$fa, \$fa, \$fa
8600	txa			9600	eor	*\$80	10600		beq	as3	11600	asc *tsx :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$ba, \$fa, \$fa, \$fa
8610	jsr	putout		9610	tax		10610		sty	t3	11610	asc *txs :.byte \$fa, \$fa, \$fa, \$fa, \$fa, \$fa, \$9a, \$fa, \$fa, \$fa
8620	jmp	ab3		9620	jmp	w2	10620		jsr	putout	11620	:
8630	ab2	lda	t5	9630	w5	sty len	10630		ldy	t3	11630	:symass messages
8640	bne	ab4		9640	cpy	*0	10640	as2	iny		11640	:
8650	ldy	t3		9650	rts		10650		cpy	len	11650	messstar .asc *symass 3.10 robert huehn feb 1986 :.byte 13,0
8660	iny:iny:iny			9660	:		10660		bne	as1	11660	messlir .byte 13:asc :first pass... :.byte 0
8670	lda	(optr).y		9670	custop = *	:custom pseudo-ops	10670	as	jmp	cp	11670	messsec .asc :second pass... :.byte 0
8680	cmp	*\$fa		9680	:		10680	as3	inc	ptr	11680	messac .byte 13:asc :assembly complete :.byte 0
8690	beq	ab4		9690	iny		10690		bne	as2	11690	messsto .byte 13:asc :symbol table overflow :.byte 0
8700	jsr	putoutop		9700	lda	(adj).y	10700		inc	ptr + 1	11700	messiq .byte 13:asc :illegal quantity :.byte 0
8710	lda	t2		9710	cmp	*\$b	10710		bne	as2	11710	messus .byte 13:asc :undefined symbol :.byte 0
8720	jsr	putout		9720	bne	cp1	10720	:			11720	messboor .byte 13:asc :branch out of range :.byte 0
8730	ab3	ldy	t3	9730	jmp	byte	10730	end	= *	:end	11730	messim .byte 13:asc :illegal mode :.byte 0
8740	dey:dey:dey			9740	cp1	cmp *\$w	10740	:			11740	messip .byte 13:asc :illegal pseudo-op :.byte 0
8750	beq	ab		9750	bne	cp2	10750		lda	flag		
8760	inc	len		9760	jmp	byte + 2	10760		bne	en		
8770	inc	len		9770	cp2	cmp *\$c6	10770		jmp	secpass		
8780	ab	jmp	next2	9780	bne	cp3	10780	en	jsr	nextword		
8790	:			9790	jmp	asc	10790		ldx	<*messac		
8800	jump = *	:jmp, jsr and jmp ()		9800	cmp	*\$80	10800		ldy	>*messac		
8810	:			9810	bne	cp4	10810		jsr	printmsg		
8820	jsr	nextword		9820	jmp	end	10820		lda	ptr		
8830	ldy	*0		9830	cp4	cmp *\$p	10830		sta	line		
8840	lda	(adj).y		9840	bne	cp5	10840		lda	ptr + 1		
8850	cmp	*\$*		9850	jmp	pad	10850		sta	line + 1		
8860	beq	jp1		9860	cp5	ldx <*messip :illegal	10860		jsr	inline		
8870	ldy	*3		9870	ldy	>*messip :pseudo-op	10870		jmp	contbas		
8880	lda	(optr).y		9880	jsr	printmsg	10880	:				
8890	jsr	putoutop		9890	jsr	inline	10890	pad	= *	:pad object with a 0 if at		
8900	jsr	eval		9900	jmp	listline	10900	:	odd byte to keep jmp tables safe			
8910	jsr	putout		9910	:		10910		lda	ptr		
8920	txa			9920	cp	lda flag	10920		and	*1		
8930	jsr	putout		9930	bne	cp6	10930		beq	pa		
8940	jmp	next2		9940	jmp	next	10940		lda	flag		
8950	jp1	inc	ad	9950	cp6	jmp next2	10950		beq	pa1		
8960	bne	jp2		9960	:		10960		lda	*0		
8970	inc	ad + 1		9970	byte	= *	10970		jsr	putout		
8980	jp2	dec	len	9980	:	:.byte and .word	10980	pa	jmp	cp		
8990	dec	len		9990	lda	*0	10990	pa1	inc	ptr		
9000	ldy	*4		10000	sta	t5	11000		bne	pa		
9010	lda	(optr).y		10010	jsr	nextword	11010		inc	ptr + 1		
9020	jsr	putoutop		10020	sty	t3	11020		bne	pa		

SYMASS Loader

Generates diskfile "symass 3.1" which you then load and run. Don't forget to save this program first.

```
PF 100 open1,8,1,"0:symass 3.1"
NM 110 print#1,chr$(1)chr$(8);
PO 120 fora = 2049to5253:readd:c = c + d
BL 130 print#1,chr$(d)::next
PF 140 close 1
NI 150 ifc<>400792thenprint " data error "
AK 160 end
MM 1000 data 11, 8, 10, 0, 158, 50, 48, 54
HO 1010 data 49, 0, 0, 0, 165, 55, 133, 40
OC 1020 data 165, 56, 133, 41, 165, 45, 133, 38
OC 1030 data 165, 46, 133, 39, 160, 0, 165, 38
KC 1040 data 208, 2, 198, 39, 198, 38, 177, 38
ND 1050 data 201, 3, 176, 79, 72, 165, 38, 208
MD 1060 data 2, 198, 39, 198, 38, 177, 38, 201
KB 1070 data 3, 144, 50, 170, 165, 38, 208, 2
IL 1080 data 198, 39, 198, 38, 177, 38, 24, 101
LK 1090 data 55, 133, 42, 138, 101, 56, 170, 104
GF 1100 data 208, 16, 165, 40, 208, 2, 198, 41
MJ 1110 data 198, 40, 138, 145, 40, 165, 42, 24
NI 1120 data 144, 10, 201, 1, 208, 4, 138, 24
HG 1130 data 144, 2, 165, 42, 72, 165, 40, 208
JE 1140 data 2, 198, 41, 198, 40, 104, 145, 40
NN 1150 data 24, 144, 163, 201, 127, 208, 237, 169
LP 1160 data 76, 141, 188, 2, 165, 40, 141, 189
DK 1170 data 2, 133, 55, 165, 41, 141, 190, 2
MC 1180 data 133, 56, 32, 99, 166, 169, 255, 133
AP 1190 data 58, 76, 188, 2, 127, 169, 0, 0
FF 1200 data 133, 2, 2, 162, 58, 255, 2, 160
PA 1210 data 58, 255, 1, 32, 150, 249, 0, 166
DD 1220 data 58, 232, 208, 3, 76, 116, 164, 162
FI 1230 data 94, 255, 2, 160, 94, 255, 1, 32
DA 1240 data 150, 249, 0, 165, 55, 133, 87, 165
DA 1250 data 56, 133, 88, 230, 122, 208, 2, 2
GC 1260 data 230, 123, 165, 122, 133, 80, 133, 78
LE 1270 data 165, 123, 133, 81, 133, 79, 32, 88
PJ 1280 data 249, 0, 208, 3, 76, 201, 245, 0
JB 1290 data 32, 38, 251, 0, 208, 7, 201, 178
KM 1300 data 208, 59, 76, 78, 246, 0, 162, 0
CL 1310 data 0, 161, 122, 201, 172, 208, 6, 32
OI 1320 data 26, 247, 0, 76, 172, 245, 0, 177
BF 1330 data 122, 201, 178, 208, 3, 76, 58, 246
NE 1340 data 0, 32, 201, 248, 0, 144, 3, 76
HD 1350 data 160, 246, 0, 160, 0, 0, 177, 122
PC 1360 data 201, 46, 208, 3, 76, 97, 251, 0
NH 1370 data 32, 95, 246, 0, 160, 0, 0, 165
JK 1380 data 89, 145, 82, 200, 165, 90, 145, 82
KK 1390 data 164, 93, 177, 122, 201, 32, 240, 7
HF 1400 data 201, 58, 240, 3, 76, 96, 245, 0
EG 1410 data 200, 152, 24, 101, 122, 133, 122, 144
JC 1420 data 2, 2, 230, 123, 76, 104, 245, 0
JJ 1430 data 230, 2, 2, 162, 109, 255, 2, 160
CH 1440 data 109, 255, 1, 32, 150, 249, 0, 165
GL 1450 data 80, 133, 78, 165, 81, 133, 79, 32
DJ 1460 data 88, 249, 0, 208, 3, 76, 37, 246
DA 1470 data 0, 32, 38, 251, 0, 240, 33, 162
PM 1480 data 0, 0, 161, 122, 201, 172, 208, 6
LN 1490 data 32, 26, 247, 0, 76, 8, 246, 0
AI 1500 data 32, 201, 248, 0, 144, 3, 76, 37
HL 1510 data 250, 0, 160, 0, 0, 177, 122, 201
GN 1520 data 46, 208, 3, 76, 97, 251, 0, 164
OB 1530 data 93, 177, 122, 201, 32, 240, 7, 201
ED 1540 data 58, 240, 3, 76, 218, 245, 0, 200
CK 1550 data 152, 24, 101, 122, 133, 122, 144, 2
NM 1560 data 2, 230, 123, 76, 226, 245, 0, 162
```

```
PN 1570 data 124, 255, 2, 160, 124, 255, 1, 32
GF 1580 data 150, 249, 0, 165, 89, 133, 57, 165
FH 1590 data 90, 133, 58, 32, 194, 189, 76, 116
LF 1600 data 164, 32, 95, 246, 0, 32, 131, 249
AE 1610 data 0, 32, 83, 247, 0, 160, 0, 0
HA 1620 data 145, 82, 200, 138, 145, 82, 76, 172
HL 1630 data 245, 0, 32, 131, 249, 0, 32, 83
IK 1640 data 247, 0, 160, 0, 0, 145, 82, 200
JK 1650 data 138, 145, 82, 76, 172, 245, 0, 165
HM 1660 data 87, 56, 233, 10, 133, 87, 176, 2
PM 1670 data 2, 198, 88, 197, 45, 165, 88, 229
BM 1680 data 46, 176, 13, 162, 143, 255, 2, 160
KJ 1690 data 143, 255, 1, 32, 150, 249, 0, 32
CP 1700 data 194, 189, 76, 71, 249, 0, 24, 165
HA 1710 data 87, 105, 8, 133, 82, 165, 88, 105
EK 1720 data 0, 0, 133, 83, 160, 8, 169, 0
PM 1730 data 0, 136, 145, 87, 208, 251, 164, 93
HB 1740 data 136, 177, 122, 145, 87, 152, 208, 248
ED 1750 data 96, 160, 0, 0, 177, 122, 201, 74
CA 1760 data 240, 19, 201, 66, 208, 22, 224, 33
AA 1770 data 240, 18, 224, 32, 240, 14, 32, 135
DA 1780 data 249, 0, 169, 2, 2, 208, 83, 32
FD 1790 data 135, 249, 0, 169, 3, 208, 76, 32
HB 1800 data 135, 249, 0, 208, 4, 169, 1, 1
LB 1810 data 208, 67, 160, 0, 0, 177, 122, 201
OC 1820 data 35, 240, 228, 201, 40, 240, 224, 164
PK 1830 data 93, 136, 240, 43, 136, 240, 40, 177
FD 1840 data 122, 201, 44, 208, 34, 200, 177, 122
HJ 1850 data 160, 7, 201, 88, 240, 1, 1, 200
HN 1860 data 177, 91, 201, 250, 240, 23, 164, 93
GD 1870 data 136, 136, 132, 93, 32, 83, 247, 0
MN 1880 data 230, 93, 230, 93, 224, 0, 0, 76
BB 1890 data 10, 247, 0, 32, 83, 247, 0, 240
OI 1900 data 171, 169, 3, 24, 101, 89, 133, 89
KJ 1910 data 144, 2, 2, 230, 90, 76, 172, 245
PG 1920 data 0, 32, 135, 249, 0, 32, 131, 249
IP 1930 data 0, 32, 83, 247, 0, 133, 89, 134
JA 1940 data 90, 96, 200, 177, 122, 133, 38, 169
MK 1950 data 0, 0, 133, 39, 200, 200, 76, 160
PJ 1960 data 247, 0, 200, 196, 93, 240, 10, 177
OJ 1970 data 122, 201, 170, 240, 4, 201, 171, 208
AO 1980 data 241, 132, 38, 32, 250, 248, 0, 164
IC 1990 data 38, 133, 38, 134, 39, 76, 160, 247
AL 2000 data 0, 169, 0, 0, 133, 40, 133, 41
HP 2010 data 133, 42, 133, 95, 160, 0, 0, 177
JD 2020 data 122, 201, 36, 208, 3, 76, 8, 248
FB 2030 data 0, 201, 34, 240, 188, 201, 172, 240
MC 2040 data 39, 201, 177, 240, 23, 201, 179, 240
MD 2050 data 19, 201, 37, 208, 3, 76, 161, 248
MN 2060 data 0, 56, 233, 48, 144, 178, 201, 10
FF 2070 data 176, 174, 76, 91, 248, 0, 133, 95
KC 2080 data 230, 122, 208, 2, 2, 230, 123, 198
OB 2090 data 93, 208, 198, 200, 165, 89, 133, 38
GG 2100 data 165, 90, 133, 39, 165, 42, 208, 11
DH 2110 data 165, 38, 133, 40, 165, 39, 133, 41
JG 2120 data 76, 210, 247, 0, 201, 170, 208, 16
BH 2130 data 24, 165, 38, 101, 40, 133, 40, 165
NJ 2140 data 39, 101, 41, 144, 18, 76, 78, 248
AJ 2150 data 0, 56, 165, 40, 229, 38, 133, 40
BA 2160 data 165, 41, 229, 39, 133, 41, 144, 124
EB 2170 data 196, 93, 240, 27, 177, 122, 133, 42
GG 2180 data 200, 152, 24, 101, 122, 133, 122, 144
DM 2190 data 2, 2, 230, 123, 56, 165, 93, 132
CP 2200 data 93, 229, 93, 133, 93, 76, 93, 247
CN 2210 data 0, 165, 95, 208, 5, 165, 40, 166
KJ 2220 data 41, 96, 201, 177, 208, 5, 165, 41
HO 2230 data 162, 0, 0, 96, 165, 40, 162, 0
AL 2240 data 0, 96, 200, 169, 0, 0, 133, 38
DI 2250 data 133, 39, 177, 122, 56, 233, 48, 144
AM 2260 data 53, 201, 10, 144, 10, 233, 17, 144
```


DG	2270 data	45, 201, 6, 176, 41, 105, 10, 6	LJ	2970 data	32, 10, 251, 0, 32, 83, 247, 0
MD	2280 data	38, 38, 39, 176, 36, 6, 38, 38	IN	2980 data	32, 27, 251, 0, 230, 93, 230, 93
DN	2290 data	39, 176, 30, 6, 38, 38, 39, 176	CN	2990 data	230, 93, 76, 8, 246, 0, 160, 0
CL	2300 data	24, 6, 38, 38, 39, 176, 18, 101	JL	3000 data	0, 177, 122, 201, 74, 208, 3, 76
FK	2310 data	38, 133, 38, 165, 39, 105, 0, 0	BK	3010 data	203, 250, 0, 201, 66, 208, 11, 224
LG	2320 data	133, 39, 176, 6, 200, 208, 196, 76	HF	3020 data	33, 240, 7, 224, 32, 240, 3, 76
KC	2330 data	160, 247, 0, 162, 166, 255, 2, 160	PL	3030 data	167, 249, 0, 32, 135, 249, 0, 208
PC	2340 data	166, 255, 1, 32, 150, 249, 0, 32	DA	3040 data	10, 160, 9, 177, 91, 32, 10, 251
OJ	2350 data	194, 189, 76, 71, 249, 0, 169, 0	FH	3050 data	0, 76, 8, 246, 0, 160, 0, 0
NB	2360 data	0, 133, 38, 133, 39, 177, 122, 56	ND	3060 data	177, 122, 201, 35, 208, 3, 76, 226
GG	2370 data	233, 48, 144, 54, 201, 10, 176, 50	IP	3070 data	249, 0, 201, 40, 208, 3, 76, 250
KC	2380 data	72, 165, 38, 166, 39, 6, 38, 38	MN	3080 data	249, 0, 201, 33, 208, 9, 230, 122
KP	2390 data	39, 176, 215, 6, 38, 38, 39, 176	ME	3090 data	208, 2, 2, 230, 123, 198, 93, 44
HO	2400 data	209, 101, 38, 133, 38, 138, 101, 39	DM	3100 data	169, 0, 0, 133, 96, 162, 3, 164
JC	2410 data	133, 39, 176, 198, 6, 38, 38, 39	NK	3110 data	93, 136, 240, 20, 136, 240, 17, 177
FM	2420 data	176, 192, 104, 101, 38, 133, 38, 165	GJ	3120 data	122, 201, 44, 208, 11, 132, 93, 232
ON	2430 data	39, 105, 0, 0, 133, 39, 176, 179	KC	3130 data	200, 177, 122, 201, 88, 240, 1, 1
DD	2440 data	200, 208, 195, 76, 160, 247, 0, 200	AD	3140 data	232, 134, 94, 32, 83, 247, 0, 240
DO	2450 data	169, 0, 0, 133, 38, 133, 39, 177	EG	3150 data	19, 164, 94, 177, 91, 32, 10, 251
GI	2460 data	122, 56, 233, 48, 144, 23, 201, 2	EN	3160 data	0, 165, 40, 32, 27, 251, 0, 138
FP	2470 data	2, 176, 19, 6, 38, 38, 39, 176	HC	3170 data	32, 27, 251, 0, 76, 189, 250, 0
PE	2480 data	149, 101, 38, 133, 38, 165, 39, 105	AM	3180 data	165, 96, 208, 233, 164, 94, 200, 200
JA	2490 data	0, 0, 133, 39, 200, 208, 226, 76	MJ	3190 data	200, 177, 91, 201, 250, 240, 222, 32
HL	2500 data	160, 247, 0, 169, 98, 252, 2, 133	BJ	3200 data	10, 251, 0, 165, 40, 32, 27, 251
NA	2510 data	91, 169, 98, 252, 1, 133, 92, 162	DE	3210 data	0, 164, 94, 136, 136, 136, 240, 4
KL	2520 data	0, 0, 160, 0, 0, 177, 91, 240	CG	3220 data	230, 93, 230, 93, 76, 8, 246, 0
JJ	2530 data	9, 209, 122, 208, 11, 200, 192, 3	HD	3230 data	32, 135, 249, 0, 160, 0, 0, 177
CG	2540 data	144, 243, 196, 93, 208, 2, 2, 56	IJ	3240 data	122, 201, 40, 240, 20, 160, 3, 177
GD	2550 data	96, 232, 165, 91, 24, 105, 13, 133	LB	3250 data	91, 32, 10, 251, 0, 32, 83, 247
JJ	2560 data	91, 144, 2, 2, 230, 92, 224, 55	AL	3260 data	0, 32, 27, 251, 0, 138, 32, 27
JD	2570 data	208, 219, 24, 96, 165, 55, 133, 82	DF	3270 data	251, 0, 76, 8, 246, 0, 230, 122
EG	2580 data	165, 56, 133, 83, 165, 82, 56, 233	DP	3280 data	208, 2, 2, 230, 123, 198, 93, 198
GK	2590 data	10, 133, 82, 176, 2, 2, 198, 83	HB	3290 data	93, 160, 4, 177, 91, 32, 10, 251
DK	2600 data	197, 87, 165, 83, 229, 88, 176, 22	CP	3300 data	0, 32, 83, 247, 0, 32, 27, 251
FL	2610 data	165, 2, 2, 208, 5, 165, 89, 166	PD	3310 data	0, 138, 32, 27, 251, 0, 230, 93
MH	2620 data	90, 96, 162, 184, 255, 2, 160, 184	BE	3320 data	76, 8, 246, 0, 201, 250, 208, 13
HE	2630 data	255, 1, 32, 150, 249, 0, 32, 194	LP	3330 data	162, 223, 255, 2, 160, 223, 255, 1
EJ	2640 data	189, 76, 71, 249, 0, 160, 0, 0	KF	3340 data	32, 150, 249, 0, 32, 194, 189, 76
GP	2650 data	177, 82, 240, 9, 209, 122, 208, 205	CF	3350 data	71, 249, 0, 160, 0, 0, 145, 89
NO	2660 data	200, 192, 8, 144, 243, 196, 38, 208	EO	3360 data	230, 89, 208, 2, 2, 230, 90, 96
KA	2670 data	196, 160, 9, 177, 82, 170, 136, 177	DD	3370 data	162, 0, 0, 160, 0, 0, 177, 122
JJ	2680 data	82, 96, 165, 57, 133, 20, 165, 58	FK	3380 data	240, 46, 201, 34, 240, 35, 224, 128
LI	2690 data	133, 21, 32, 19, 166, 32, 201, 166	DF	3390 data	240, 16, 201, 58, 240, 34, 201, 59
FP	2700 data	76, 116, 164, 165, 78, 133, 122, 165	GM	3400 data	240, 30, 201, 178, 240, 26, 201, 32
DM	2710 data	79, 133, 123, 160, 0, 0, 177, 122	BD	3410 data	240, 3, 200, 208, 225, 192, 0, 0
CA	2720 data	133, 78, 200, 177, 122, 133, 79, 240	MF	3420 data	208, 15, 230, 122, 208, 217, 230, 123
JN	2730 data	21, 200, 177, 122, 133, 57, 200, 177	BB	3430 data	208, 213, 138, 73, 128, 170, 76, 70
PL	2740 data	122, 133, 58, 24, 165, 122, 105, 4	IH	3440 data	251, 0, 132, 93, 192, 0, 0, 96
FK	2750 data	133, 122, 144, 2, 2, 230, 123, 96	CM	3450 data	200, 177, 122, 201, 66, 208, 3, 76
FO	2760 data	164, 93, 200, 44, 164, 93, 152, 24	HK	3460 data	158, 251, 0, 201, 87, 208, 3, 76
JL	2770 data	101, 122, 133, 122, 144, 2, 2, 230	MH	3470 data	160, 251, 0, 201, 198, 208, 3, 76
JO	2780 data	123, 76, 38, 251, 0, 134, 38, 132	NO	3480 data	3, 252, 0, 201, 128, 208, 3, 76
CD	2790 data	39, 160, 0, 0, 177, 38, 240, 6	HC	3490 data	41, 252, 0, 201, 80, 208, 3, 76
OD	2800 data	32, 210, 255, 200, 208, 246, 96, 160	IJ	3500 data	72, 252, 0, 162, 237, 255, 2, 160
GA	2810 data	3, 177, 91, 32, 10, 251, 0, 32	JL	3510 data	237, 255, 1, 32, 150, 249, 0, 32
JJ	2820 data	135, 249, 0, 32, 83, 247, 0, 56	CD	3520 data	194, 189, 76, 71, 249, 0, 165, 2
NO	2830 data	233, 1, 1, 176, 1, 1, 202, 56	FC	3530 data	2, 208, 3, 76, 172, 245, 0, 76
OM	2840 data	229, 89, 133, 38, 138, 229, 90, 170	HN	3540 data	8, 246, 0, 169, 0, 0, 133, 96
IE	2850 data	24, 165, 38, 105, 128, 138, 105, 0	CD	3550 data	32, 135, 249, 0, 132, 94, 160, 0
MM	2860 data	0, 240, 13, 162, 202, 255, 2, 160	GI	3560 data	0, 177, 122, 201, 44, 240, 5, 200
IC	2870 data	202, 255, 1, 32, 150, 249, 0, 32	EE	3570 data	196, 94, 208, 245, 165, 2, 2, 240
FJ	2880 data	194, 189, 76, 71, 249, 0, 165, 38	FB	3580 data	51, 132, 93, 200, 165, 94, 132, 94
GE	2890 data	32, 27, 251, 0, 76, 8, 246, 0	LE	3590 data	56, 229, 94, 133, 94, 176, 4, 169
KE	2900 data	230, 122, 208, 2, 2, 230, 123, 160	LO	3600 data	0, 0, 133, 94, 32, 83, 247, 0
MI	2910 data	10, 177, 91, 32, 10, 251, 0, 198	IM	3610 data	32, 27, 251, 0, 165, 96, 240, 4
IO	2920 data	93, 32, 83, 247, 0, 32, 27, 251	FC	3620 data	138, 32, 27, 251, 0, 165, 94, 240
IP	2930 data	0, 76, 8, 246, 0, 230, 122, 208	BJ	3630 data	36, 164, 93, 200, 152, 24, 101, 122
KK	2940 data	2, 2, 230, 123, 165, 93, 56, 233	OF	3640 data	133, 122, 144, 192, 230, 123, 208, 188
LG	2950 data	4, 168, 132, 93, 177, 122, 160, 11	FH	3650 data	24, 165, 96, 240, 2, 2, 169, 1
DM	2960 data	201, 44, 240, 1, 1, 200, 177, 91	ND	3660 data	1, 105, 1, 1, 101, 89, 133, 89

FK	3670 data 144, 2, 2, 230, 90, 196, 94, 208	KH	4370 data 66, 82, 75, 250, 250, 250, 250, 250
NM	3680 data 175, 76, 148, 251, 0, 32, 135, 249	KG	4380 data 250, 0, 0, 250, 250, 250, 66, 86
IJ	3690 data 0, 160, 1, 1, 177, 122, 201, 34	OP	4390 data 67, 80, 250, 250, 250, 250, 250, 250
NK	3700 data 240, 16, 166, 2, 2, 240, 15, 132	AN	4400 data 250, 250, 250, 66, 86, 83, 112, 250
KK	3710 data 94, 32, 27, 251, 0, 164, 94, 200	IH	4410 data 250, 250, 250, 250, 250, 250, 250, 250
PO	3720 data 196, 93, 208, 234, 76, 148, 251, 0	PL	4420 data 67, 76, 68, 250, 250, 250, 250, 250
FO	3730 data 230, 89, 208, 244, 230, 90, 208, 240	LA	4430 data 250, 216, 250, 250, 250, 67, 76, 73
EA	3740 data 165, 2, 2, 208, 3, 76, 201, 245	BI	4440 data 250, 250, 250, 250, 250, 250, 88, 250
NI	3750 data 0, 32, 135, 249, 0, 162, 124, 255	NP	4450 data 250, 250, 67, 76, 86, 250, 250, 250
MI	3760 data 2, 160, 124, 255, 1, 32, 150, 249	LJ	4460 data 250, 250, 250, 184, 250, 250, 250, 67
EL	3770 data 0, 165, 89, 133, 57, 165, 90, 133	OG	4470 data 80, 89, 204, 250, 250, 196, 250, 250
KG	3780 data 58, 32, 194, 189, 76, 174, 167, 165	JE	4480 data 250, 192, 250, 250, 73, 78, 88, 250
MP	3790 data 89, 41, 1, 1, 240, 9, 165, 2	CM	4490 data 250, 250, 250, 250, 250, 232, 250, 250
IO	3800 data 2, 240, 8, 169, 0, 0, 32, 27	OA	4500 data 250, 76, 83, 82, 78, 94, 250, 70
BA	3810 data 251, 0, 76, 148, 251, 0, 230, 89	NM	4510 data 86, 250, 74, 250, 250, 250, 78, 79
GL	3820 data 208, 249, 230, 90, 208, 245, 76, 68	JK	4520 data 80, 250, 250, 250, 250, 250, 250, 234
BH	3830 data 65, 173, 189, 185, 165, 181, 250, 250	ID	4530 data 250, 250, 250, 80, 72, 65, 250, 250
JL	3840 data 169, 161, 177, 83, 84, 65, 141, 157	EM	4540 data 250, 250, 250, 250, 72, 250, 250, 250
KD	3850 data 153, 133, 149, 250, 250, 250, 129, 145	BB	4550 data 80, 72, 80, 250, 250, 250, 250, 250
OJ	3860 data 66, 78, 69, 208, 250, 250, 250, 250	AL	4560 data 250, 8, 250, 250, 250, 80, 76, 65
FO	3870 data 250, 250, 250, 250, 250, 66, 69, 81	FA	4570 data 250, 250, 250, 250, 250, 250, 104, 250
CG	3880 data 240, 250, 250, 250, 250, 250, 250, 250	NF	4580 data 250, 250, 80, 76, 80, 250, 250, 250
CN	3890 data 250, 250, 67, 77, 80, 205, 221, 217	FC	4590 data 250, 250, 250, 40, 250, 250, 250, 82
OE	3900 data 197, 213, 250, 250, 201, 193, 209, 74	CH	4600 data 79, 76, 46, 62, 250, 38, 54, 250
OJ	3910 data 83, 82, 32, 250, 250, 250, 250, 250	EC	4610 data 42, 250, 250, 250, 82, 176, 0, 0
KB	3920 data 250, 250, 250, 250, 76, 68, 88, 174	NP	4620 data 110, 126, 250, 102, 118, 250, 106, 250
KL	3930 data 250, 190, 166, 250, 182, 250, 162, 250	IJ	4630 data 250, 250, 82, 84, 73, 250, 250, 250
GN	3940 data 250, 82, 84, 83, 250, 250, 250, 250	AH	4640 data 250, 250, 250, 64, 250, 250, 250, 83
KD	3950 data 250, 250, 96, 250, 250, 250, 76, 68	LB	4650 data 69, 67, 250, 250, 250, 250, 250, 250
ON	3960 data 89, 172, 188, 250, 164, 180, 250, 250	BJ	4660 data 56, 250, 250, 250, 83, 69, 68, 250
DO	3970 data 160, 250, 250, 66, 77, 73, 48, 250	AI	4670 data 250, 250, 250, 250, 250, 248, 250, 250
KM	3980 data 250, 250, 250, 250, 250, 250, 250, 250	IM	4680 data 250, 83, 69, 73, 250, 250, 250, 250
GD	3990 data 68, 69, 67, 206, 222, 250, 198, 214	EK	4690 data 250, 250, 120, 250, 250, 250, 84, 65
MM	4000 data 250, 250, 250, 250, 250, 175, 0, 0	IJ	4700 data 89, 250, 250, 250, 250, 250, 250, 168
PJ	4010 data 0, 0, 45, 61, 57, 37, 53, 250	JA	4710 data 250, 250, 250, 84, 83, 88, 250, 250
LB	4020 data 250, 41, 33, 49, 66, 67, 83, 176	NL	4720 data 250, 250, 250, 250, 186, 250, 250, 250
MP	4030 data 250, 250, 250, 250, 250, 250, 250, 250	PO	4730 data 84, 88, 83, 250, 250, 250, 250, 250
PF	4040 data 250, 73, 78, 67, 238, 254, 250, 230	HE	4740 data 250, 154, 250, 250, 250, 18, 83, 89
ND	4050 data 246, 250, 250, 250, 250, 250, 66, 67	AP	4750 data 77, 65, 83, 83, 32, 51, 46, 49
EP	4060 data 67, 144, 250, 250, 250, 250, 250, 250	AC	4760 data 48, 32, 82, 79, 66, 69, 82, 84
DJ	4070 data 250, 250, 250, 84, 89, 65, 250, 250	OP	4770 data 32, 72, 85, 69, 72, 78, 32, 70
NC	4080 data 250, 250, 250, 250, 152, 250, 250, 250	JB	4780 data 69, 66, 32, 49, 57, 56, 54, 13
PD	4090 data 66, 80, 76, 16, 250, 250, 250, 250	EP	4790 data 0, 0, 13, 70, 73, 82, 83, 84
CM	4100 data 250, 250, 250, 250, 250, 65, 83, 76	GB	4800 data 32, 80, 65, 83, 83, 46, 46, 46
KP	4110 data 14, 30, 250, 6, 22, 250, 10, 250	JF	4810 data 0, 0, 83, 69, 67, 79, 78, 68
DL	4120 data 250, 250, 67, 76, 67, 250, 250, 250	KC	4820 data 32, 80, 65, 83, 83, 46, 46, 46
IG	4130 data 250, 250, 250, 24, 250, 250, 250, 65	EE	4830 data 0, 0, 13, 65, 83, 83, 69, 77
BD	4140 data 68, 67, 109, 125, 121, 101, 117, 250	EI	4840 data 66, 76, 89, 32, 67, 79, 77, 80
OH	4150 data 250, 105, 97, 113, 69, 176, 0, 0	BF	4850 data 76, 69, 84, 69, 0, 0, 13, 83
CN	4160 data 77, 93, 89, 69, 85, 250, 250, 73	GJ	4860 data 89, 77, 66, 79, 76, 32, 84, 65
MH	4170 data 65, 81, 84, 88, 65, 250, 250, 250	JK	4870 data 66, 76, 69, 32, 79, 86, 69, 82
DH	4180 data 250, 250, 250, 138, 250, 250, 250, 67	NF	4880 data 70, 76, 79, 87, 0, 0, 13, 73
IF	4190 data 80, 88, 236, 250, 250, 228, 250, 250	IK	4890 data 76, 76, 69, 71, 65, 76, 32, 81
AM	4200 data 250, 224, 250, 250, 74, 77, 80, 76	LG	4900 data 85, 65, 78, 84, 73, 84, 89, 0
BK	4210 data 108, 250, 250, 250, 250, 250, 250, 250	CH	4910 data 0, 13, 85, 78, 68, 69, 70, 73
PP	4220 data 250, 84, 65, 88, 250, 250, 250, 250	IN	4920 data 78, 69, 68, 32, 83, 89, 77, 66
AP	4230 data 250, 250, 170, 250, 250, 250, 73, 78	NK	4930 data 79, 76, 0, 0, 13, 66, 82, 65
MJ	4240 data 89, 250, 250, 250, 250, 250, 250, 200	PM	4940 data 78, 67, 72, 32, 79, 85, 84, 32
JD	4250 data 250, 250, 250, 83, 84, 89, 140, 250	KN	4950 data 79, 70, 32, 82, 65, 78, 71, 69
KN	4260 data 250, 132, 148, 250, 250, 250, 250, 250	EL	4960 data 0, 0, 13, 73, 76, 76, 69, 71
KO	4270 data 176, 65, 0, 0, 13, 29, 25, 5	NL	4970 data 65, 76, 32, 77, 79, 68, 69, 0
DM	4280 data 21, 250, 250, 9, 1, 1, 17, 68	MK	4980 data 0, 13, 73, 76, 76, 69, 71, 65
NL	4290 data 69, 89, 250, 250, 250, 250, 250, 250	AB	4990 data 76, 32, 80, 83, 69, 85, 68, 79
DJ	4300 data 136, 250, 250, 250, 68, 69, 88, 250	LF	5000 data 45, 79, 80, 0, 0
CA	4310 data 250, 250, 250, 250, 250, 202, 250, 250		
BG	4320 data 250, 83, 84, 88, 142, 250, 250, 134		
HE	4330 data 250, 150, 250, 250, 250, 250, 83, 66		
AH	4340 data 67, 237, 253, 249, 229, 245, 250, 250		
CE	4350 data 233, 225, 241, 66, 73, 84, 44, 250		
LA	4360 data 250, 36, 250, 250, 250, 250, 250, 250		

News BRK

Submitting NEWS BRK Press Releases

If you have a press release which you would like to submit for the NEWS BRK column, make sure that the computer or device for which the product is intended is prominently noted. We receive hundreds of press releases for each issue, and ones whose intended readership is not clear must unfortunately go straight to the trash bin. It should also be mentioned here that we only print product releases which are in some way Applicable to Commodore equipment.

Transactor News

Transactor on Microfiche

We now have 18 Transactor Magazines on microfiche! – all of Volume 4, Volume 5, and Volume 6. According to Computrex, our fiche manufacturer, the strips are the “popular 98 page size”, so they should be compatible with every fiche reader.

To keep things simple, we're making the price of the fiche the same as magazines, with one exception. A single back issue will be \$4.50 (remember, you can now get those 5 Transactors that are no longer available on paper!), and subscriptions will also be the same price as shown on the order card. The exception? A complete set of 18 (Volumes 4, 5, and 6) will cost just \$39.95!

Transactor Mail Order News

Our mail-order department is expanding nicely, but our mail-order card isn't. Seems we just can't find any more room to put more text without making it so small that you can't read it. So, if you're using the card to order, we suggest you pull it out and cross-reference with the list below for more details.

■ Inner Space Anthology \$14.95

This is our ever popular Complete Commodore Inner Space Anthology. Even after a year, we still get inquiries about its contents. Briefly, The Anthology is a reference book – it has no “reading” material (ie. “paragraphs”). In 122 pages, there are memory maps for 5 CBM computers, 3 Disk Drives, and maps of COMAL; summaries of BASIC commands, Assembler and MLM commands, and Wordprocessor and Spreadsheet commands. Machine Language codes and modes are summarized, as well as entry points to ROM routines. There are sections on Music, Graphics, Network and BBS phone numbers, Computer Clubs, Hardware, unit-to-unit conversions, plus much more. . . about 2.5 million characters total!

■ The Toolbox (PAL and POWER) \$79.95

PAL and POWER from Pro-Line are two of the most popular programs for the Commodore 64. PAL is an easy-to-use assembler (most assembler listings in The Transactor are in PAL format), and POWER is a programmer's aid package that adds editing features and useful commands to the programming environment. They come with two nice manuals, and our price is \$50 less than suggested retail!

■ The GLINK C64 to IEEE Interface \$49.95

The GLINK plugs into the cartridge port, but doesn't extend the port for more cartridges (for that you'll need a “motherboard” of some kind). The other side of the GLINK is a IEEE card-edge suitable for a PET-IEEE cable. From there, any IEEE device can be accessed including disk drives, modems, printers, etc. The GLINK is “transparent” – that means it won't interfere with programs, except those that rely on the serial routines which it replaces (ie. programs with built-in “fastloaders” for the 1541 won't like the presence of the GLINK). It has no manual (aside from one page of installation instructions) because it alters nothing and leaves everything unchanged! An on-board switch allows you to select Serial or IEEE. GLINK works with both the C64 and the C128 in 64 mode.

■ The TransBASIC Disk \$9.95

This is the complete collection of every TransBASIC module ever published. There are over 120 commands at your disposal. You pick the ones you want to use, and in any combination! It's so simple that a summary of instructions fits right on the disk label. The manual describes each of the commands, plus how to make your own commands.

■ Jim Butterfield's 1986 Diary \$5.95 (plus 50¢ p&h)

Jim has put together a handy pocket reference that includes the most-used areas of memory maps, command summaries, equipment summaries, some short programs, sound and video, machine language, and a glossary, followed by a pocket diary and a neat colour map of the London England Underground, in case you're going there.

■ The SM Compiler \$39.95 US, \$49.95 Cdn

This compiler is for BASIC 7.0 on the Commodore 128. We've compared it with two others, and this is the one we like. Watch for that comparison in an upcoming issue.

■ Super Kit 1541 \$29.95 US, \$39.95 Cdn

Super Kit is, quite simply, the best disk file utility there is. No more losing those valuable copy-protected originals (like what's happened to me twice in the last month). See the News BRK item ahead.

■ Paperback Writer C64 \$39.95 US, \$49.95 Cdn

■ Paperback Planner C64 \$39.95 US, \$49.95 Cdn

■ Paperback Filer C64 \$39.95 US, \$49.95 Cdn

■ Paperback Writer C128 \$49.95 US, \$69.95 Cdn

■ Paperback Planner C128 \$49.95 US, \$69.95 Cdn

■ Paperback Filer C128 \$49.95 US, \$69.95 Cdn

■ Paperback Dictionary \$14.95 US, \$19.95 Cdn

In our opinion, the Paperback packages from Digital Solutions are the best you can get on their own – the fact that they work with each other makes them even better. Planner and Filer data can be loaded into the Writer, Writer text can be sent to the Filer, and etcetera. The Dictionary spell checker works with both versions of the Writer.

As mentioned earlier, all issues of The Transactor from Volume 4 Issue 01 forward are now available on microfiche. Some issue are ONLY available on microfiche – these are marked “MF only”. This list also shows the “themes” of each issue. “Theme issues” didn't start until Volume 5, Issue 01.

■ Vol. 4, Issue 01 (■ Disk 1) ■ Vol. 4, Issue 04 – MF only (■ Disk 1)

■ Vol. 4, Issue 02 (■ Disk 1) ■ Vol. 4, Issue 05 – MF only (■ Disk 1)

■ Vol. 4, Issue 03 (■ Disk 1) ■ Vol. 4, Issue 06 – MF only (■ Disk 1)

■ Vol. 5, Issue 01 – Sound and Graphics (■ Disk 2)

■ Vol. 5, Issue 02 – Transition to Machine Language (■ Disk 2)

■ Vol. 5, Issue 03 – Piracy and Protection – MF only (■ Disk 2)

■ Vol. 5, Issue 04 – Business & Education – MF only (■ Disk 3)

■ Vol. 5, Issue 05 – Hardware & Peripherals (■ Disk 4)

■ Vol. 5, Issue 06 – Aids & Utilities (■ Disk 5)

■ Vol. 6, Issue 01 – More Aids & Utilities (■ Disk 6)

■ Vol. 6, Issue 02 – Networking & Communications (■ Disk 7)

■ Vol. 6, Issue 03 – The Languages (■ Disk 8)

■ Vol. 6, Issue 04 – Implementing The Sciences (■ Disk 9)

■ Vol. 6, Issue 05 – Hardware & Software Interfacing (■ Disk 10)

■ Vol. 6, Issue 06 – Real Life Applications (■ Disk 11)

■ Vol. 7, Issue 01 – ROM / Kernel Routines (■ Disk 12)

Notes: The Transactor Disk #1 contains all program from Volume 4, and Disk #2 contains all programs from Volume 5, Issues 1-3. Afterwards there is a separate disk for each issue. Disk 8 from The Languages Issue contains COMAL

0.14, a soft-loaded, slightly scaled down version of the COMAL 2.0 cartridge. And Volume 6, Issue 05 published the directories Transactor Disks 1 to 9.

The Viewtron Starter Kit

Since Viewtron is now shipping starter kits for free (\$2.50 US. p&h), we've discontinued distribution. See the ad this issue for more details.

Transactor Open On Viewtron!

Remember, any of the above items can be ordered from our Transactor Section on Viewtron. Just sign on, enter "transactor", and proceed to the order section. We'll respond to confirm, and usually have your order out the same week. See the "Viewtron Keywords" article on page 26 of this issue for more info.

In the next Transactor we'll have a complete rundown on using the Transactor section, which, for the most part, will apply to just about any Viewtron section. If you get on before that, leave us some mail - we'll be happy to hear from you!

The Transactor Communications Disk

We're currently working on a "Transactor Communications Disk". We already have permission from Viewtron to include their software and hope to include many more. When finished it could host as many as 15 different modem programs and may even require two diskettes. We also plan an "all-in-one" manual to go with it so you'll never be without a telecommunications program for virtually any host computer and protocol. But it's not ready yet so don't send any orders. More next issue.

Industry News

Workshops In Computer-Assisted Instruction In Music

The lab for Computer-Assisted Instruction In Music at Brooklyn College will be offering two workshops this summer for music teachers who are interested in using computers as a teaching tool. Each workshop will last five days and include 15 hours of classroom instruction on the Commodore 64 computers and their applications in teaching music. The cost: \$200.00/workshop. The dates are July 7-11, 1986 and July 14-18, 1986.

These workshops are offered in affiliation with the Center for Computer Music at Brooklyn College. For further information and application, contact:

Gary S. Karpinski, Director
Lab for CAI in Music
Conservatory of Music
Brooklyn College, Brooklyn, NY 11210
(718) 780-5286

Distressed Commodore Users Hotline

On January 1st, 1986, David Bradley began operating a brand new Freeware service for new Commodore computer users. It is a hotline for users to call when they are having trouble(s) with their new machines. The hotline operates Monday to Friday from 2:00 PM to 10:00 PM and the number for users to call is (416) 488-4776. Users that want more information about the service can write to:

Distressed Commodore Users Hotline
147 Roe Avenue
Toronto, Ontario, Canada
M5M 2H8

Or they can call any line of the Bradley Brothers Bulletin Board System at (416) 487-5833, (416) 481-8661, (416) 481-9047 or (416) 277-9991. All four lines operate 24 hours a day, 7 days a week.

The 1541 Revealed

"The 1541 Revealed" is a 48-page booklet from Write Protect Publishing. Written by Felix Rivera and Evelio Quiros, the book contains information and diagrams concerning the 1541 disk drive's internals, and practical tips to prolong the unit's life. Sections of the book include: An overview of the 1541, how and why problems arise, "The Naked 1541", Cleaning, lubricating, adjusting and aligning. A section on modifications explains how to: change device numbers, add a front-mounted power switch, change the head end-stop to a "springy" one, and add a write-protect switch. The writing style is informal and easy to understand.

A labelled general board layout of the 1541 is found in the centre of the booklet, and a "track checker" program and a list of references is included at the end. Price of the booklet is \$5.00. For more information, contact:

Write Protect Publishing Company
Suite 4E, 135 Charles Street
New York, NY 10014

Used Computer Listing Service

Due to widespread demand, Comp-Used, which has helped buyers and sellers of used computer equipment in the North East for two years, is expanding its services.

Comp-Used is a listing service that facilitates the sale and purchase of used computer equipment. Anyone with equipment worth over \$100 can contact the Comp-Used computer to register the product for sale. In the same vein, anyone in the market to purchase equipment can call the Comp-Used computer for information. Comp-Used connects the buyer and seller and they finalize the sale. When a transaction takes place, the seller pays Comp-Used a small commission; there is no charge to buy.

To talk with the Comp-Used telephone computer, call (203) 762-8677

Comp-Used
85 Rivergate Drive
Wilson, CT 06897

Steve Jobs and Pixar Employees Buy Pixar

San Raphael, CA. -- Pixar, the computer graphics division of Lucasfilm Ltd., has announced that it has been acquired by Steven P. Jobs and the employees of Pixar. Pixar, now an independent company, will design, manufacture and market high performance computers and software specifically tailored for state of the art computer graphics and image processing applications.

The new firm has a product, the Pixar Image Computer, ready for market. Developed during the last three years at Lucasfilm Ltd., the Pixar Image Computer is nearly 200 times faster than conventional minicomputers at performing complex graphic and image computations. At these specialized tasks, the Pixar Image Computer is also faster than a \$6 million supercomputer. The Pixar Image Computer will be introduced to the commercial and scientific markets within the next 90 days and will sell for approximately \$125,000.00.

Pixar was originally formed in 1979 by George Lucas to bring high technology to the film industry. Lucasfilm Ltd. will continue to use the Pixar Image Computer and other technologies to produce computer animation for films through its special effects division, Industrial Light & Magic (ILM), and for home entertainment through its Games Group.

MSD Disk Drive Information Exchange

Now that Micro System Development, the maker of MSD Disk Drives, is no longer in business, an information exchange is being set up to serve the needs of MSD disk drive users. The first project is a database of compatible software.

Users of MSD disk drives are encouraged to participate in the exchange.

The MSD Information Exchange is a no fees, not-for-profit, user service. Those who contribute information to the exchange will be provided the following services:

1. For a self-addressed stamped envelope, a printout of available information in one selected category.
2. For a blank disk with mailer and return postage, a copy of the Information Exchange data disk in Superbase 64 format (data disk only).

Typical entries in the exchange data base include--

Word Processor, PaperClip 64, Batteries Included, 64C Edition
SD-2: Fully compatible serial or parallel (Quicksilver interface)

Spreadsheet, Multiplan 64, Hesware, v. 1.06
SD-2: Partially compatible. Data files may not be saved.

Backup, MSD Shure Copy, Megasoft
SD-2: Serial compatible. Parallel: Incompatible with Quicksilver

Entertainment, Flight Simulator II, Sub Logic
SD-2: Incompatible

The information exchange will also maintain files on the availability of technical information on MSD disk drives including parts, service, service or maintenance manuals, wiring diagrams, memory maps, etc. as provided by users.

Software News

Introducing Super Kit/1541

Prism Software is proud to introduce Super Kit/1541 for the Commodore 64. Super Kit is the most full-featured 1541 utility package to be found today. Just look at the features offered:

Single/Dual Normal Copier: Copies a disk with no errors in 32.68 seconds. Dual version has graphics and music.

Single/Dual Nibble Copier: Nibble copies a disk in 34.92 seconds. Dual version has graphics and music.

Single/Dual File Copier: 6 times normal DOS speed. Includes multi-copy, multi-scratch, view-edit BAM, and new Super DOS Mode.

Track and Sector Editor: Full editing of t&s in hex, dec, ascii, bin. Includes monitor/disassembler with printout commands.

GCR Editor: Yes disk fans, a full blown sector by sector or track by track GCR editor. Includes Bit Density Scan.

Super DOS I: Fast boot for Super DOS. 150 blocks in 10.12 seconds.

Super DOS II: Screen on and still loads 150 blocks in 14.87 seconds.

Super Nibbler: Quite frankly, if it can be copied on a 1541, this will do it! Including Abacus, Timeworks, Accolayde, Epyx, Acti-vision, Electronic Arts.

The price, \$29.95 plus \$3.00 shipping and handling.

Prism Software
401 Lake Air Drive, Suite D
Waco, Texas 76710
Orders (817) 757-4031 (or use order card at center)
Tech (817) 751-0200

DISKORGANIZER For The C64

You probably bought your computer, at least partially, to help you get organized. And you probably started with a handful of disks on which you stored all your files. But now you have boxes and boxes of disks with directories that look like they were organized by a not particularly bright chimpanzee. You like elegance and order, and you wish you could organize your disks, but this seems such a gargantuan task that you keep putting it off. The order of the files on a Commodore disk directory seems to be engraved in stone. (The same stone holds the header.) The only way to reorganize the directory is through laborious file copying to a fresh disk, right? Wrong! We have good news: DISKORGANIZER for the C-64.

With this ultimate disk utility for the C-64 you can quickly and easily sort and rearrange the disk directory of any unprotected disk to meet your own specifications, and the new directory is actually written back onto the disk! Using a convenient screen editor you can also change the header, scratch files, copy files of any size to another disk, rename files, add 'fences' to mark off sections of the disk for easy reading and independent sorting, 'scratch-protect' any file, position individual files anywhere in the directory, and, of course, print out copies of your revised directories.

You may have a copier utility or utility to rename the header or you may use the wedge for common disk commands. But you don't have a single program that will take care of all your disk housekeeping (even housekeeping you didn't think possible) quickly and easily. But you will, if you get DISKORGANIZER and get organized. DISKORGANIZER is available for \$29.95 from:

The G.A.S.S. Company
970 Copeland
North Bay, Ontario, Canada
P1B-3E4
(705) 474-9602

Amiga Spreadsheet, Telecommunications and BBS

Micro-Systems Software Inc. has released three new software tools for the Amiga. The first is a spreadsheet called Analyse!. Similar in concept to Lotus 1-2-3, the \$99.95 program takes maximum advantage of Amiga's capabilities (pull down menus, mouse, workbench) and can produce professional sized spreadsheets (256 columns x 8,156 rows).

The second package, Online!, is a full-featured telecommunications system for the Amiga that retails for \$69.95. The third package, BBS-PC, is a versatile electronic bulletin board system that transforms any Amiga into an online information network.

The \$99.95 program easily interfaces to a hard disk or keeps up with a 2400 bps modem. In addition, BBS-PC works in the "background", so the Amiga can answer the phone and take messages while users are working on other projects.

All three packages are being distributed by Softeam, National Software Distributors and Computer Software Services in the U.S., and in Canada by Phase 4 Distributors. For additional information contact:

Brown-Wagh Publishing
100 Verona Court
Los Gatos, California 95030 (408) 395-3838

The Sourcerer 6500 Series Disassembler

The Sourcerer is a multi-pass disassembler which converts 6500 series machine language (object code) into Assembly Language (source code). It operates disk-to-disk, disk-to-screen, or disk-to-printer. The commented Assembly Language which is produced can be immediately re-assembled with the Commodore assembler, or loaded for editing with the Commodore editor. Any specified

range of code within a program can be disassembled. Long programs automatically produce linked disk files for easy editing. All addresses referenced in the code are converted to labels, in several sorted categories. The Sourcerer is written in 100% machine language for fast operation, and will disassemble a 20k program with several thousand address labels in less than 13 minutes. The time required for the final output of source code depends on the speed of the output device (disk, screen, or printer).

The Sourcerer is only \$29.95 ppd. on a 1541 disk complete with operating instructions. Order from:

Chessoft Ltd.
723 Barton Street
Mt. Vernon, IL 62864

Help Master For The Commodore 64

Help Master 64 is a software/book package that will aid Basic programmers. Help Master 64 provides instant, on-line help screens on each and every Basic command used by the Commodore 64 computer.

Once loaded, Help Master 64 remains hidden in memory until you need it. It takes up absolutely none of the Basic RAM, is completely compatible with the DOS wedge, and has no effect on your ability to write, edit, load, save or run any Basic program.

When you need help, typing the quote mark plus the name of the command will instantly produce a half-screen overlay showing the Commodore abbreviation for the command, the proper syntax, a description of the command and reference page numbers in various manuals which will provide more information than is available on-screen. This half-screen format will allow you to view both your actual program line and the Help Master example at the same time so you can see what the differences are.

After viewing the help screen, you may restore the information that was on the top half of the screen, or you can correct your program line while the help screen is still being displayed.

Help Master 64 comes with the 'Handbook Of Basic for the Commodore 64', an excellent 368 page reference manual on Commodore Basic by Frederick E. Mosher and David I. Schneider, published by Bradey Communications, Inc. The package has a retail price of \$29.95. For more information contact:

Master Software
6 Hillery Court
Randallstown, MD 21133 (301) 922-2962

Hardware News

RESWITCH from Compusave

Reswitch is a reset switch/power-on indicator for the Commodore 64 which replaces the existing power-on LED. Installation requires no drilling or cutting, as the unit pops into the same hole as the existing LED. The Reswitch is a transparent pushbutton containing an LED and acts exactly like the original LED except that pushing down on it causes the 64 to reset. The package comes with detailed installation instructions and everything needed to hook up. Price is \$10.00. Contact:

Comp U Save
115 Essex St. Suite #146
New York, NY 10002

Uninterruptible Power Supply

An on-line, sine wave Uninterruptible Power Supply is being introduced by Electronic Specialists. Capable of supplying up to 20 minutes power during

extended power outages, the on-line unit operates without disruptive switching transients. Automatic internal battery recharge is incorporated.

Wide band EMI/RFI filtering and High-Speed, High-Current Spike Suppression provide extended protection. Added protection is provided by an integral over-load/short-circuit proof configuration.

A front panel TEST switch permits convenient power removal to check front panel monitors and complete system operation.

Line phase lock, automatic Blackout illumination, Battery-Saver automatic shut-down option and external battery option are featured. Available in 250 and 500 watts. For more information, contact:

Electronic Specialists Inc.
171 South Main Street
Natick, Massachusetts
01760 1-800-225-4876

80 Column Mono Cable For The C-128

This is the cable for an 80 column monochrome display as described in many Commodore specific magazine. It eliminates the need for an RGB monitor and allows the use of any composite color or monochrome monitor. Excellent for data base and word processing applications. (See next item for address)

40/80 Column Switch Cable For The C-128

A flip of a switch on the connector is all that's needed to change from 40 to 80 column display and back again. Plus a simple keystroke (ESC X). In 40 column mode all 16 colors are available on your color monitor.

It's small and easy to install, with no bulky switches, boxes or exposed components.

The 80 Column Mono Cable retails for \$9.95. The 40/80 Column Switchable cable retails for \$23.95. For more information contact:

Innovative Computer Accessories
1249 Downing Street, PO Box 789
Imperial Beach, CA 92032-0837 (619) 224-1177

C.A.S.E.
COMMODORE ASSOCIATION SOUTH/EAST

presents

THE COMMODORE SHOW

at the

Opryland Hotel

Nashville, Tennessee
April 26th & 27th, 9:00 a.m. to 5:00 p.m.

NATIONAL COMMODORE SPEAKERS!
VENDORS AND DISPLAYS!
SHOW SPECIALS AND DOOR PRIZES!
SEE INNOVATIONS AVAILABLE FOR THE COMMODORE MARKET!
THE ONLY COMMODORE CONFERENCE IN THE SOUTH EAST!
USER GROUP ORIENTED!

2-DAY REGISTRATION FEE: \$10.00 (\$7.50 before April 15)

*Tickets are available at all affiliated clubs,
or for more information, contact:*

C.A.S.E. • P.O. BOX 110386 • NASHVILLE, TN 37222
(615) 834-5679 • (615) 834-2073 • (205) 854-3496

**PAYS
\$40**

per page for articles

We're also looking for
professionally
drawn cartoons!

Send all material to:

The Editor
The Transactor
500 Steeles Avenue
Milton, Ontario
L9T 3P7

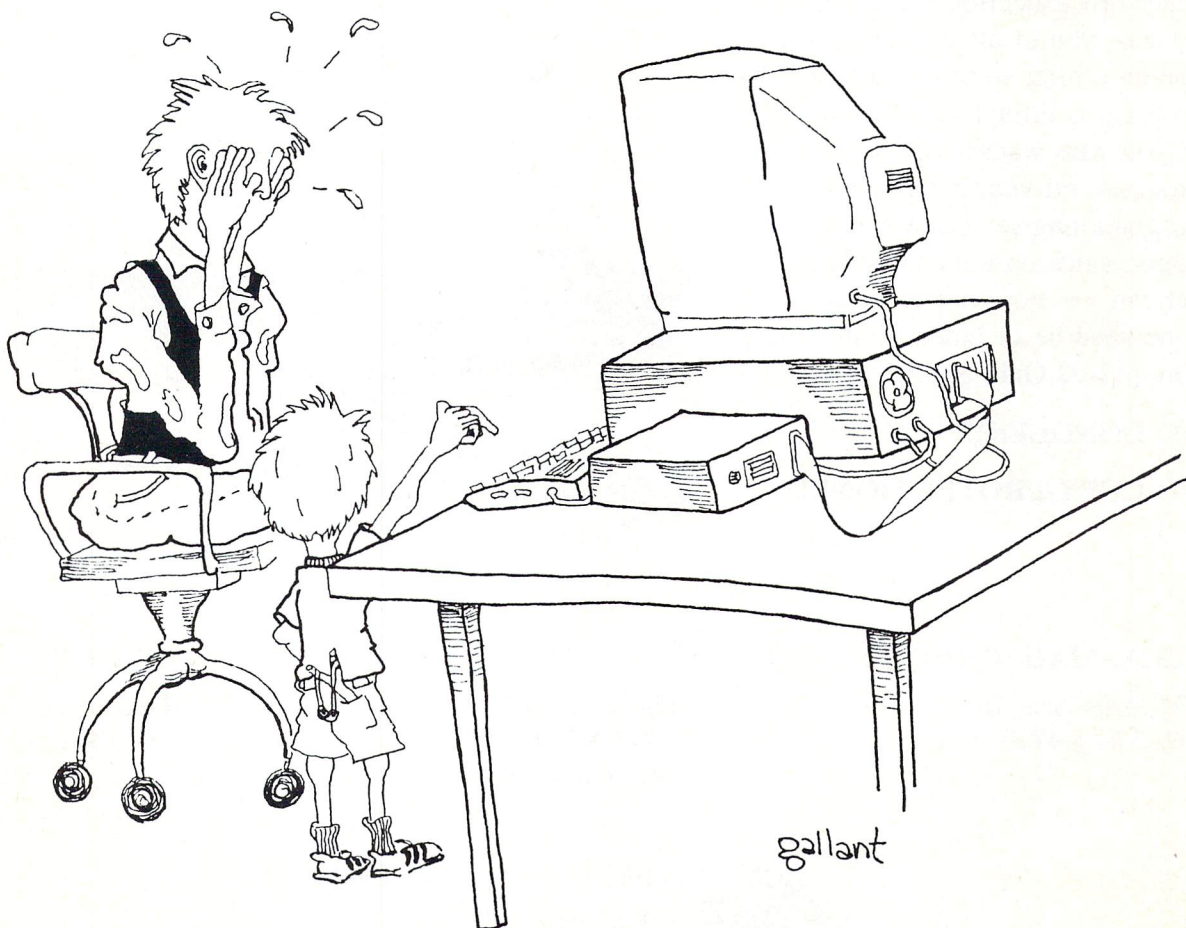
Volume 6 Editorial Schedule

Issue#	Theme	Copy Due	Printed	Release Date
1	More Aids & Utilities	Feb 1	Mar 22	April 1/85
2	Communications & Networking	Apr 1	May 24	June 1
3	Languages	Jun 1	Jul 26	August 1
4	Implementing The Sciences	Aug 1	Sep 20	October 1
5	Hardware & Software Interfacing	Oct 1	Nov 22	December 1
6	Real Life Applications	Dec 1	Jan 24	February 1/86

Volume 7 Editorial Schedule

1	ROM Routines / Kernel Routines	Feb 1	Mar 21	April 1
2	Games From The Inside Out	Apr 1	May 23	June 1
3	Programming The Chips	Jun 1	Jul 25	August 1
4	Gadgets and Gizmos	Aug 1	Sep 26	October 1
5	Simulations and Modelling	Oct 1	Nov 21	December 1
6	Programming Techniques	Dec 1	Jan 23	February 1/87

Advertisers and Authors should have material submitted no
later than the 'Copy Due' date to be included
with the respective issue.



THAT'S HOW YOU DO IT DADDY!



While driving deep into the Black Forest of Germany, our slow and unreliable Volkswagen started sputtering and puttering and then to our dismay, just quit running. After hours of unrelentlous tinkering, our poor little mobile was running, slowly, but running. And we wanted to get out of that dank and dark forest quickly. Befuddled and confused, we were ecstatic to see a strange little Gnome emerge from behind a tree. This creature, who called himself Hacker, used his infinite wisdom and wizardry to fix our Volkswagen and get us speedily on our way. Well, we were so impressed with Hacker Gnome's wizardry, that we convinced him to reveal his secrets for speed and reliability. **And we are passing these secrets along to you so that you can write the very best Basic Programs.**

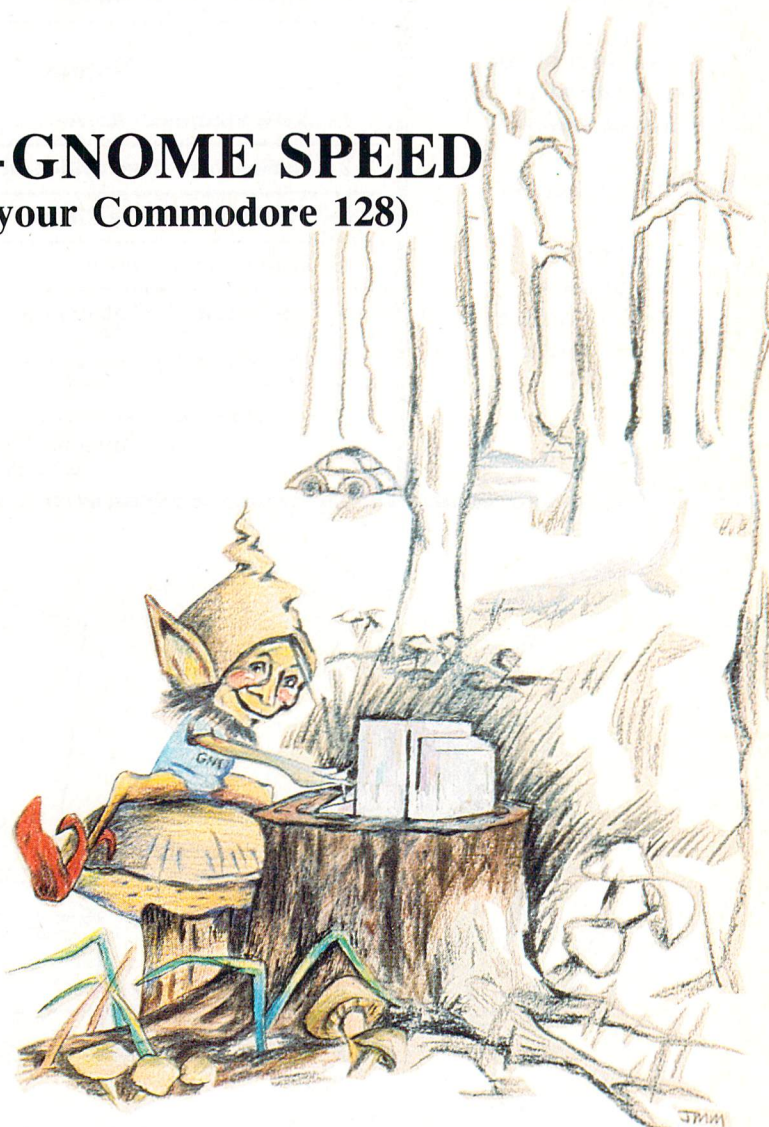
INTRODUCING-GNOME SPEED

(A Basic 7.0 Compiler for your Commodore 128)

Gnome Speed allows you to transform virtually any Basic 7.0 Program into a compiled version that is as sophisticated and fast as if it were written in machine code. Simply compiling your program with **Gnome Speed** not only gives you super-fast execution speed, but also informs you of all your program coding errors, so that your compiled program is error-free. And for those of you who want to sell your program, all your efforts and programming secrets will remain yours, since only the compiled version — not your Basic source code need be included on the disk. **The price?** Only \$59.95 (U.S.)

NO DONGLES !!

NO COPY-PROTECTION!!!



U.S.A. MAIL ORDERS:

SM Software, Inc.
1-215-682-4920

CANADIAN MAIL ORDERS:

The Transactor
1-416-878-8438
(see order card)

DEALER INQUIRIES:

Micro-Pace, Inc.
1-217-356-1884



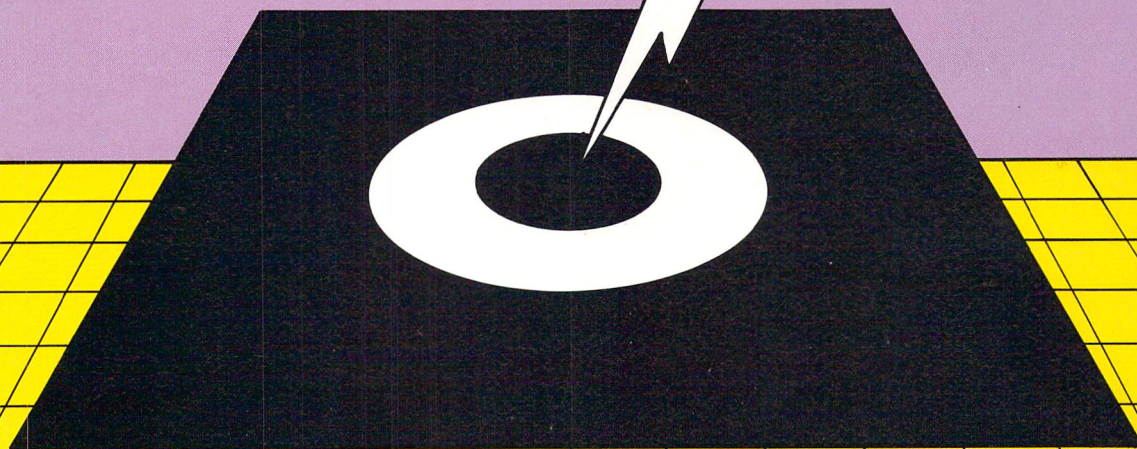
SM SOFTWARE, INC.

P.O. Box 27
Mertztown, PA. 19539-0027
1-215-682-4920

I N T R O D U C I N G

SUPER KIT 1541

Has it all!



BY MARTY FRANZ & JOE PETER

SINGLE/DUAL NORMAL COPIER

Copies a disk with no errors in 32.68 seconds. Dual version has graphics & music.

SINGLE/DUAL NIBBLE COPIER

Nibble Copies a disk in 34.92 seconds. Dual version has graphics & music.

SINGLE/DUAL FILE COPIER

7 times normal DOS speed. Includes multi-copy, multi-scratch, view/edit BAM, & NEW SUPER DOS MODE. In Super DOS Mode, it transfers 7-15 times normal speed, copies 150 blocks in 23 seconds.

TRACK & SECTOR EDITOR

Full editing of t&s in hex, dec, ascii, bin. Includes monitor/disassembler with printout commands.

GCR EDITOR

Yes disk fans, a full blown sector by sector or track by track GCR Editor. Includes TRUE Bit Density/Track Scan.

3 SUPER DOS FAST LOADERS

Over 15 times normal DOS speed. Super DOS Files are still Commodore DOS compatible. Imagine loading 150 blocks in 10 seconds.

SUPER NIBBLER/ SUPER DISK SURGEON

Quite frankly, these will provide you the user with the backup you need! Even copies itself.

\$29.95 U.S.

PLUS \$3.00 SHIPPING/HANDLING CHARGE — \$5.00 C.O.D. CHARGE

PRISM
SOFTWARE

401 LAKE AIR DR., SUITE D • WACO, TEXAS 76710
ORDERS (817) 757-4031 • TECH (817) 751-0200
MASTERCARD & VISA ACCEPTED

SUPER KIT/1541 is for archival use only! We do not condone nor encourage piracy of any kind.

See center page for mail order card.

THE TIME SAVER



J. MOSTACCI

Type in a lot of Transactor programs?
Does the above time and appearance of the sky look familiar?
With The Transactor Disk, any program is just a LOAD away!

Only \$7.95 Per Issue
6 Disk Subscription (one year)
Just \$45.00
(see order form at center fold)